AUGUST 1984

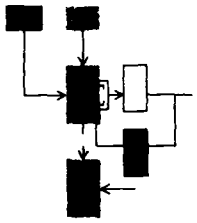
LIDS-TH-1396

(12)

Research Supported By:

Defense Advanced Research Projects Agency
Contract N00014-84-K-0357

National Science Foundation
Contract NSF-ECS-8310698



THE DESIGN AND PERFORMANCE ANALYSIS OF AN ARBITER FOR A MULTI-PROCESSOR SHARED-MEMORY SYSTEM

Shahrukh S. Merchant

Laboratory for Information and Decision Systems
MASSACHUSETTS INSTITUTE OF TECHNOLOGY, CAMBRIDGE, MASSACHUSETTS 02139

August 1984 LIDS-TH-1396

THE DESIGN AND PERFORMANCE ANALYSIS OF AN ARBITER FOR A MULTI-PROCESSOR SHARED-MEMORY SYSTEM

by

Shahrukh S. Merchant

This report is based on the unaltered thesis of Shahrukh S. Merchant submitted in partial fulfillment of the requirements for the degree of Master of Science at the Massachusetts Institute of Technology in September 1984. This research was conducted at the M.I.T. Laboratory for Information and Decision Systems with partial support provided by the Defense Advanced Research Projects Agency under Contract N00014-84-K-0357 and by the National Science Foundation under Contract NSF-ECS-8310698.

Laboratory for Information and Decision Systems
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) UNCLASSIFIED

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A146074	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE DESIGN AND PERFORMANCE ANALYSIS OF AN ARBITER FOR A MULTI-PROCESSOR SHARED-MEMORY SYSTEM		5. TYPE OF REPORT & PERIOD COVERED Thesis
		6. PERFORMING ORG. REPORT NUMBER LIDS-TH-1396
7. AUTHOR(s) Shahrukh S. Merchant		8. CONTRACT OR GRANT NUMBER(s) DARPA Order No. 3045/2-2-84 Amendment #11 ONR/N00014-84-K-0357
9. PERFORMING ORGANIZATION NAME AND ADDRESS Massachusetts Institute of Technology Laboratory for Information and Decision Systems Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Program Code No. 5T10 ONR Identifying No. 049-383
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, Virginia 22209		12. REPORT DATE August 1984
		13. NUMBER OF PAGES 104
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Program Code 437 Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release: distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A "round-robin" arbiter has been designed and implemented for sharing memory in a multi-processor system. A study is made, via analysis and simulation, of the performance of this arbiter under various load conditions. In particular, distributions are obtained for the waiting time of an arriving customer under various load conditions, and expressions are obtained for the idle time distribution and the mean busy time.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. The effect of a heavy or light user in the midst of other moderate users is also examined, as is the effect of longer guaranteed idle periods by individual customers after a service completion.

Many of the results obtained with a round-robin arbiter are compared with those obtained using a First-come, First-served arbiter. In the course of doing this, a simple relation has been found between the waiting time distribution and the distribution of the number of customers in any discrete-time First-come, First-served system with a single work-conserving deterministic server.

The Design and Performance Analysis of an Arbiter for a Multi-Processor Shared-Memory System

by

Shahrukh S. Merchant

B.S., University of Wisconsin, Madison, 1981

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements of
the Degree of

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

September 1984

Signature of Author Shahrukh S. Merchant
Department of Electrical Engineering and
Computer Science, 24 July, 1984

Certified by _____
Prof. R.G. Gallager, Thesis Supervisor

Accepted by _____
Chairman, Electrical Engr. and Computer Science

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

Figures

Tables

ABSTRACT

A "round-robin" arbiter has been designed and implemented for sharing memory in a multi-processor system. A study is made, via analysis and simulation, of the performance of this arbiter under various load conditions. In particular, distributions are obtained for the waiting time of an arriving customer under various load conditions, and expressions are obtained for the idle time distribution and the mean busy time.

The effect of a heavy or light user in the midst of other moderate users is also examined, as is the effect of longer guaranteed idle periods by individual customers after a service completion.

Many of the results obtained with a round-robin arbiter are compared with those obtained using a First-come, First-served arbiter. In the course of doing this, a simple relation has been found between the waiting time distribution and the distribution of the number of customers in any discrete-time First-come, First-served system with a single work-conserving deterministic server.

ACKNOWLEDGEMENTS

This work was supported in part by grants from the National Science Foundation (NSF-ECS-8310698) and the Defense Advanced Research Projects Agency (N00014-84-K-0357).

I would like to extend my appreciation to my thesis advisor and graduate counsellor, Prof. Robert Gallager for his constant interest and supportive attitude, and to Mr. Daniel Helman for his many helpful ideas and for the fruitful discussions I have had with him.

PART I

DESIGN AND IMPLEMENTATION

1. INTRODUCTION TO CLUMPS

The focus of this thesis is the design and analysis of the arbiter used in the Communications LangUage Multi-Processor System (CLUMPS). The purpose of this section is to acquaint the reader with the basic structure of (CLUMPS), so as to provide some perspective for the role of the arbiter in the system. This section is, therefore, global and brief in its description of the CLUMPS system; the reader is referred to [HEL84] for details.

1.1. Purpose and Philosophy of CLUMPS

The Communications LangUage Multi-Processor System (CLUMPS) is a project at the Laboratory for Information and Decision Systems (LIDS) for building a truly distributed Computer Communications Network [GAL83, HEL82, HEL84, TUB83]. A typical network is shown schematically in Figure 1.1.

The system consists of several nodes, connected together in an arbitrary topology via branches. The system is truly distributed in the sense that all its nodes are heirarchically equal; there is no master-slave relationship.(1)

---

(1) The "host" connected to a node in Figure 1.1 exists only for the purpose of initializing the system and collecting performance data. Its existence does not imply that the node to which it is connected has a

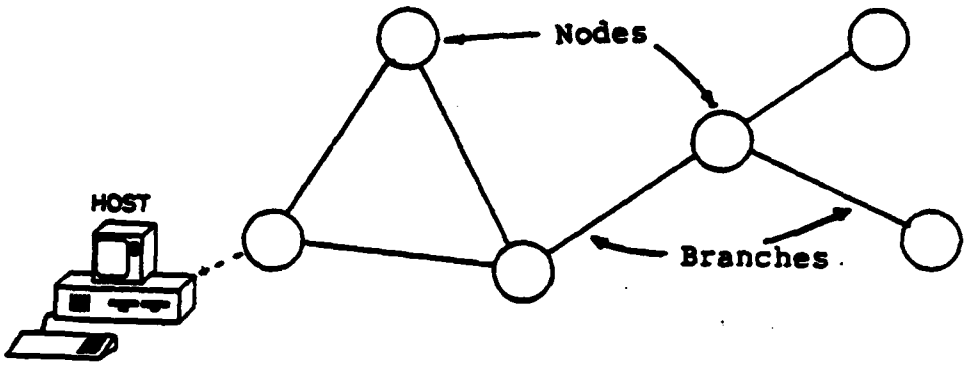


Figure 1.1. A configuration of CLUMPS nodes.

The network is to be used in the real-time simulation of network control algorithms such as those that control flow and congestion in networks, including, but not restricted to, communication networks. The idea is that one can effect faster, more realistic simulations on a real network than by simulating the network entirely via software on a computer. Additional details on the philosophy of CLUMPS may be found in [HEL84].

---

privileged status.

1.2. Outline of CLUMPS architecture

1.2.1. Architecture of a node

Each node shown in Figure 1.1 contains processing, storage and communications capabilities. As shown in Figure 1.2, each node can have up to eight branches, with which it

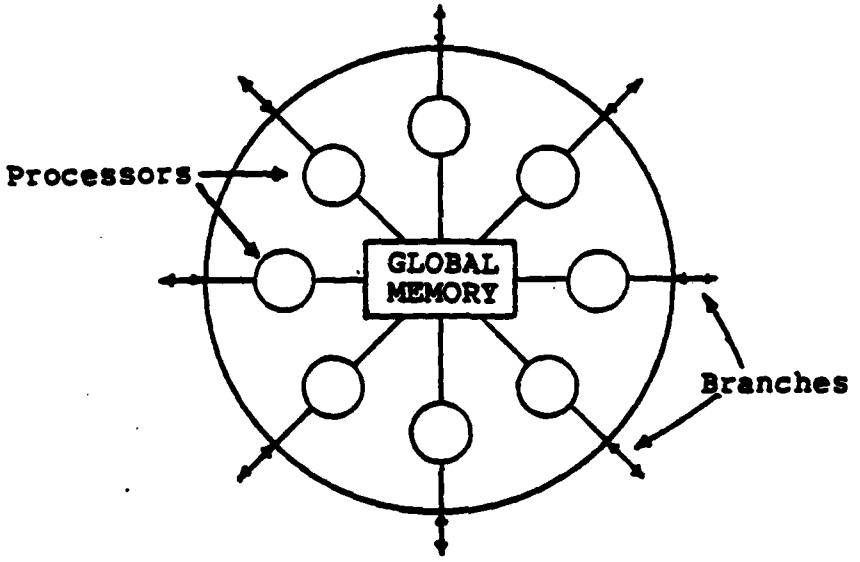


Figure 1.2. Structure of a CLUMPS node

may be connected to other nodes. The processing power of a node is contained in eight microprocessors, one associated with each branch. The processors communicate with each other only through a region of global memory, shared by all the processors in a node. These processors are closely coupled, in that computational tasks required of a node are allocated to idle processors by the operating system in a way that is transparent to any process outside the node. For all practical purposes, therefore, one may think of the processing power of a node as residing in a single

processing unit. Note, however, that there is no central processor that has a distinct master role in controlling the other processors within a node. The CLUMPS system is thus distributed in two independent (and unrelated) ways: firstly, processing is distributed at the network level (all nodes are "equal") and, secondly, at the node level (all processors within a node are "equal").

See Appendix A, [HEL83] and [MER82] for additional details on the CLUMPS architecture.

1.2.2. Overall architecture

Each branch in a node is implemented as a serial communication line. A 64 x 64 electronic crossbar switch [LEB84] permits the configuration of arbitrary topologies with up to 64 branches.

The host, shown in Figure 1.1, is an IBM Personal Computer, responsible for loading the operating system software into all the nodes in the network upon power-up or system reset. It may be programmed to receive performance and statistical data on the algorithms being simulated on the network.

1.2.3. Software

The primary language supported by the system is the high-level language NIL (Network Interface Language) [PAR82, BUR82, NIK84], developed at the IBM Research Center in Yorktown Heights, New York. This language was chosen

because of its suitability for distributed multiprocessor systems for communications applications.

The NIL compiler [HEL84, NIK84] does not generate native machine code for the branch processors (which are Motorola 6809 processors [MOT81]); it generates an intermediate sequence of "pseudo-opcodes" which are interpreted at run-time by an interpreter that is resident at each processor. Although this 2-stage approach results in slower execution, it was chosen because it results in very compact code and facilitates sharing of processes. This, in turn, leads to fewer global memory cycles which could otherwise be a potential bottleneck. It is also easier to implement and test and results in a more transportable compiler.

A distributed operating system runs at each node to allocate or queue tasks to any of the eight processors in a way that is transparent to the application program. See [HEL84] for details on the functioning of the operating system.

1.3. Functions of the global memory unit

The "global memory unit" could, more accurately, be called a "global functions unit" for, in addition to (*a*) read/write memory, it also contains (*b*) a programmable counter/timer, (c) circuitry to allow any of the microprocessors in a node to identify the node as well as the branch (within the node) with which it is associated and

(d) logic to allow any branch to reset the node or to generate an interrupt to all processors within a node.

The read/write memory is, however, the primary function of the global memory unit. It is used by the processors comprising a branch in two ways: 1) to store the intermediate compiled NIL code for interpretation by the interpreter and 2) to store data packets en route to or from other nodes.

Appendix A contains a complete schematic diagram for the entire global memory unit, including a memory map, timing diagrams and a description of circuit operation.

2. ROLE OF ARBITER IN GLOBAL MEMORY UNIT

2.1. Function of arbiter

As stated earlier, the global memory unit is shared by up to eight processors, only one of which has access to it at any given time. The function of the arbiter is to (a) accept requests for access to the global memory from the processors, (b) decide which one gets access for the next cycle in the event that two or more processors have requests pending and (c) signal its decision to the processor chosen.

2.2. Functional specifications of arbiter

2.2.1. Interface between global memory board and branches

In addition to the common bus signals shared by the global memory and branch boards, each branch has a pair of request-acknowledge lines connecting it to the arbiter on the global memory board. A branch processor makes a request by asserting its active-low request line to the logic 0 state. The request line stays in this state until the arbiter grants the request by asserting the corresponding acknowledge line (driving it to logic 0). Upon receiving the acknowledge signal, the requesting device removes its request and proceeds to use the following memory cycle for a read or write. The acknowledge stays active (low) for the duration of the cycle (a constant 400 *ns* at the clock rate used).

For convenience in identifying devices (branch

processors), we will number them from 0 to 7, and refer to their request and acknowledge signals as Req_i and Ack_i respectively, for i in the range of 0 to 7.

2.2.2. Performance considerations

An important requirement of the arbiter is that it be fair; all eight devices should receive equal priority service, on the average. The reason for this is that all branches of a node should perform identically; the hardware should not introduce a bias caused by asymmetric performance of the branches.

Furthermore, due to restrictions on how long memory cycles of a processor can be "stretched" [MOT81], the waiting time of a device must be bounded. A straightforward way to achieve this is to require that while any branch has a request pending, no other branch can have more than one request serviced. This guarantees that the waiting time for any device is at most eight cycles.

As a request must be latched and stable well before the start of a memory cycle which is affected by that request, sufficient "pipelining" of requests (about half a cycle) should be provided so that there is no "dead time" between memory cycles to process requests. For reasons of

efficiency, we will also require that the arbiter not be idle as long as there are requests pending.(1)

2.3. The "round-robin" solution

The above requirements can be met by a so-called round-robin arbiter. This arbiter serves device #0, device #1 and so on, up to device #7, and then returns to serving device #0. Of course, those devices not making requests are skipped.

Another way of describing this strategy is as follows: If device i was the last one served, then, for the next cycle, device (i+1) mod 8 has the highest priority, device (i+2) mod 8 the next highest priority, and so on, with device (i+8) mod 8 (which is just device i) having the lowest priority.

The equation describing the next-cycle status of each of the 8 acknowledgement signals, Ack_i', as a function of current requests, Req_i, and current acknowledgement signals, Ack_i is:

---

(1) An exception to this is when cycles are "stolen" occasionally for refreshing the dynamic memory. Section A.3 has details.

$$Ack_i' = Req_i\,[Ack_{i-1} +$$
$$Ack_{i-2}\overline{Req}_{i-1} +$$
$$Ack_{i-3}\overline{Req}_{i-2}\overline{Req}_{i-1} +$$
$$\ldots +$$
$$Ack_i\overline{Req}_{i-7}\overline{Req}_{i-6}\overline{Req}_{i-5}\overline{Req}_{i-4}\overline{Req}_{i-3}\overline{Req}_{i-2}\overline{Req}_{i-1}]$$

where the subscripts are all modulo 8. This may be written in the less transparent but more compact form:

$$Ack_i' = Req_i \bigvee_{j=1}^{8} [Ack_{i-j} \left(\bigwedge_{k=1}^{j-1} \overline{Req}_{i-k}\right)] \qquad (2.1)$$

If the Ack_is are viewed as being the state of the arbiter, this is the equation that relates the new state to the present state and inputs.(1)

2.3.1. Comments on fairness

As originally conceived, designed and built, the arbiter had an inherent source of unfairness, which was thought to be necessary to simplify the design so that it could be implemented in its entirety on a single Programmable Array Logic [NAT82] device. To be specific, when no requests are pending (system idle), all the

---

(1) These equations, while correct in principle, cannot be implemented in the above form. Slight modifications have to be made to handle the case where there are no requests (who gets the following cycle?) and to ensure that the arbiter is initialized to a legal state upon power-up. These details are relegated to Section A.5 in Appendix A.

acknowledgement lines, which also serve to retain the arbiter state, are inactive. Thus, should two or more devices simultaneously make a request in a subsequent cycle, one cannot tell which one should get served first based on the number of the last served device, because this information has not been retained. The solution, in the original version of the arbiter, was to simply give lower-numbered devices priority in this case, and was chosen because it was easily implemented.

It was believed that this would have a negligible effect on fairness because it only happens when (*a*) the system is idle and (*b*) two or more devices make requests simultaneously. If (*a*) occurs with high probability then the system is very lightly loaded and (*b*) will occur with low probability. The hypothesis that this would not have a significant effect on fairness was tested via simulation. The results, shown in Section 3.6, reveal that in the case of all devices having equal frequency of requests, there could be significant unfairness between the most favoured device (#0) and the least favoured device (#7), as measured by the difference between the mean waiting times for these devices. At very low and very high loading, the difference was a few percentage points and could be considered small. However, for the intermediate values of loading most likely to be encountered in the system, the difference in mean waiting times between devices 0 and 7 was higher than 20% in

some cases.

This was deemed unacceptable as it could seriously skew the results of algorithm simulations run on CLUMPS, so it was decided to modify the arbiter design to make it truly fair. Details are in Section A.5; basically the modification consists of changing the state description of the arbiter so that it reflects the last device served (however long before) rather than just the device served, if any, in the previous cycle. The increase in complexity of the design was, fortunately, much smaller than originally believed.

In the remainder of this document, the original arbiter design shall be referred to as the biased arbiter and the modified design the fair arbiter, if it is necessary to distinguish between the two and the distinction is not clear from context.

2.3.2. Rejection of a First-come First-served (FCFS) arbiter

As shown in Section 3.4, an arbiter which processes requests in a first-come first-served manner has a more regular behaviour than a round-robin arbiter in that the waiting time distribution of the former has a smaller variance than that of a round-robin arbiter (although they both have the same mean waiting time). In the case where all devices are generating requests with approximately equal frequency, it seems an intuitively reasonable method to use.

The major reason for rejecting a FCFS arbiter is that

it is considerably more complicated to implement. It requires more memory than the round-robin arbiter as it needs to maintain ranks for all devices with active requests. This could, in the worst case, require the arbiter to retain ordering information for eight previous cycles.

This additional complexity is reflected in the number of states required to implement the arbiter. For the round-robin arbiter, we require only 8 states to retain the identity of the last served device. For the FCFS arbiter, we would require 8! = 40320 states (the number of ways that the eight devices can be ranked) to retain the current ranks of all 8 devices. A FCFS arbiter could not have been implemented on a single programmable chip as was the round-robin arbiter.

The resolution of simultaneous arrivals is also a more difficult problem in a FCFS arbiter. For the round-robin arbiter, this problem arose only when the system was idle, and was fairly easy to solve, as described in Sections 2.3.1 and A.5.

In the FCFS arbiter, one would, in effect, need a secondary round-robin (or similar) scheme to resolve simultaneous arrivals.(1)

(1) Another common problem with FCFS systems in general is that a single very heavy user can monopolize the resources of the system; round-robin type solutions are usually superior in this respect. However, since a device in our system cannot generate a request until after its previous request has been serviced, this is not as severe a problem here. Results of simulations presented in Section 4 show that if mean waiting time is used as a measure of fairness, a round-robin arbiter is more "fair" than a FCFS one. For the FCFS arbiter, a heavy user could get as much as a 25% advantage in waiting time over the other users. While hardly astronomical, this difference could be significant in some applications.

PART II

PERFORMANCE ANALYSIS AND SIMULATIONS

3. THE BASIC SYMMETRIC MODEL

3.1. Setting up a simplified model

To obtain useful results, it is desirable to model this system as a Markov process; since the process is a discrete-time one, we will model it as a discrete-time, discrete-state Markov chain. To preserve the Markov property without introducing a plethora of states, we assume that for every cycle or time slot in which a device is not active, there is a constant probability that it will generate a request, independent of past history and of other devices. The one exception is that a device cannot make a request in the cycle immediately following one in which it had access to the memory. This concession to reality can fortunately be made without complicating the model unduly; it is realistic because in practice the requests are "pipelined" by about half a cycle, and since a device cannot generate a request until after it has completed the current memory cycle, it cannot generate another request in time to get the following cycle. Furthermore, to simplify the analysis, we assume that this constant probability of generating a request, which we shall call p, is the same for all devices. All of Section 3 assumes that we are using the fair arbiter (cf. Section 2.3.1).

3.1.1. Nomemclature

In the terminology of queueing theory, requests by the

devices (processors) will be referred to as customer arrivals. The granting of access to memory by the arbiter and the subsequent completion of that memory cycle will be referred to as a service completion. Note that we have a single server system that is work conserving (since the server is always busy when there is work to be done). The service is deterministic, i.e., it takes a constant time of one unit. The arrivals are not memoryless since they depend on the state of the system.

Similar nomenclature will be used in the rest of this thesis.

3.1.2. Modelling the system exactly

Modelling the system completely requires a total of about 2^{11} or 2048 states (2^8 = 256 states representing the status--active or inactive--of the 8 input lines, times 8 for each of the 8 devices which may currently have priority). Even allowing for the fact that all of these states are not valid, there are far too many states to obtain any useful analytic results. While there is some symmetry in the system, it does not seem to permit substantial simplification.

Therefore, it was decided to simulate the system to obtain statistics on

a) the steady-state distribution, $\underline{x}$, of the number of customers and

b) the steady-state distribution, $\underline{w}$, of the waiting

time of customers.

This simulation was performed on an IBM Personal Computer. The simulation results are given in Section 3.4 and a Pascal listing of the program appears in Appendix B.2. For reference, the modelling of the complete system will be referred to as the round-robin model.

3.1.3. Modelling as a First-come, First-served (FCFS) system

The round-robin discipline of the arbiter is not first-come, first-served, but there are advantages to modelling the system as such. Specifically, the identities of the customers in the system are no longer important; for the purpose of computing the distribution of the number of customers and waiting times, it is sufficient to keep track of only the number of customers (0-8) in the system. Thus, a 9-state Markov process is sufficient to model the system, and the problem is analytically tractable. For reference, this model will be referred to as the FCFS model.

In spite of the tremendous simplification resulting from the FCFS assumption, one can still get useful results. Specifically, since the server is work-conserving and all customers are identical, it must be true that

a) the steady-state distribution of the number of customers in the FCFS and round-robin models are equal, since the number of customers in the system and the arrival rates do not depend upon the service discipline, and

b) the mean waiting time in the two models are the same. Furthermore, one would expect, intuitively, that the variance of the waiting time in the round-robin model would be larger than that in the FCFS model, since the latter is, in a sense, more "regular."

3.1.4. Results to be sought

In Section 3, we will determine the distributions of the number of customers in the system (Section 3.2) and waiting time (Section 3.3) for the FCFS model and compare these to those obtained via simulation for the actual round-robin model (Section 3.4). Expressions will also be derived for the distribution of idle periods in the system and the mean length of a busy period (Section 3.5). Finally, we will compare the fairness characteristics of the "biased" and "fair" arbiters (Section 3.6).

The only variable parameter will be p, the probability of an inactive customer making a request.

3.2. Distribution of the number of customers

Figure 3.1 shows the Markov chain associated with the

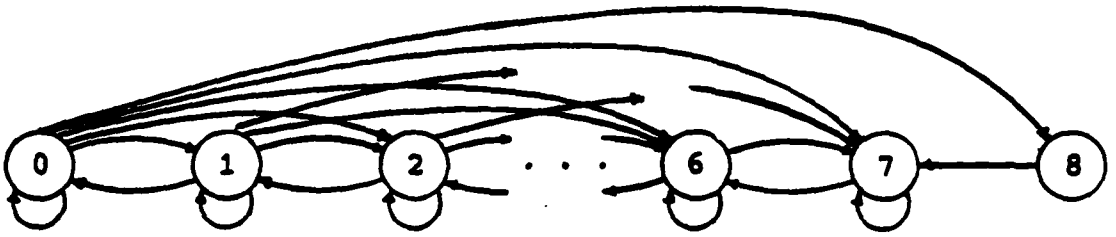


Figure 3.1. Markov chain for FCFS model

FCFS model. Let $P(i,j)$ = Single step probability of going from state i to state j. For generality in the derivations, we assume that the number of requesting devices is N, instead of 8.

Then,

$P(0,j)$ = Pr (j of N devices make a request)

$$= \binom{N}{j} p^j (1-p)^{N-j} \qquad \text{(for } j = 0,1,\ldots,N\text{)}.$$

For $i = 1,2,\ldots,N$,

$P(i,j)$ = Pr (j-i+1 of N-i devices make requests)

$$= \binom{N-i}{j-i+1} p^{j-i+1}(1-p)^{N-j-1} \quad \text{(for } j = i-1,\ldots,N-1\text{)}$$

(= 0 for $j = 0,\ldots,i-2$ and $j=N$).

For any given value of $p \in [0,1]$, we can solve for the steady-state probability vector $\underline{\pi} = \underline{\pi}P$, where P is the matrix [P(i,j)] and it is understood that $\underline{\pi}$ and P are functions of p.

A Pascal program that does this calculation for any given $p \in [0,1]$ is shown in Appendix B. Figure 3.2 shows this distribution for various values of p. Note that $\pi(8)$ is always negligibly small; this is because one can get to state 8 only from state 0 and only if all devices make a request simultaneously, i.e.,

$$\pi(8) = \pi(0)\, p^8.$$

For this to be significant, p must be close to 1, in which case we would expect $\pi(0)$ to be very small; in any event, $\pi(8)$ is always small.

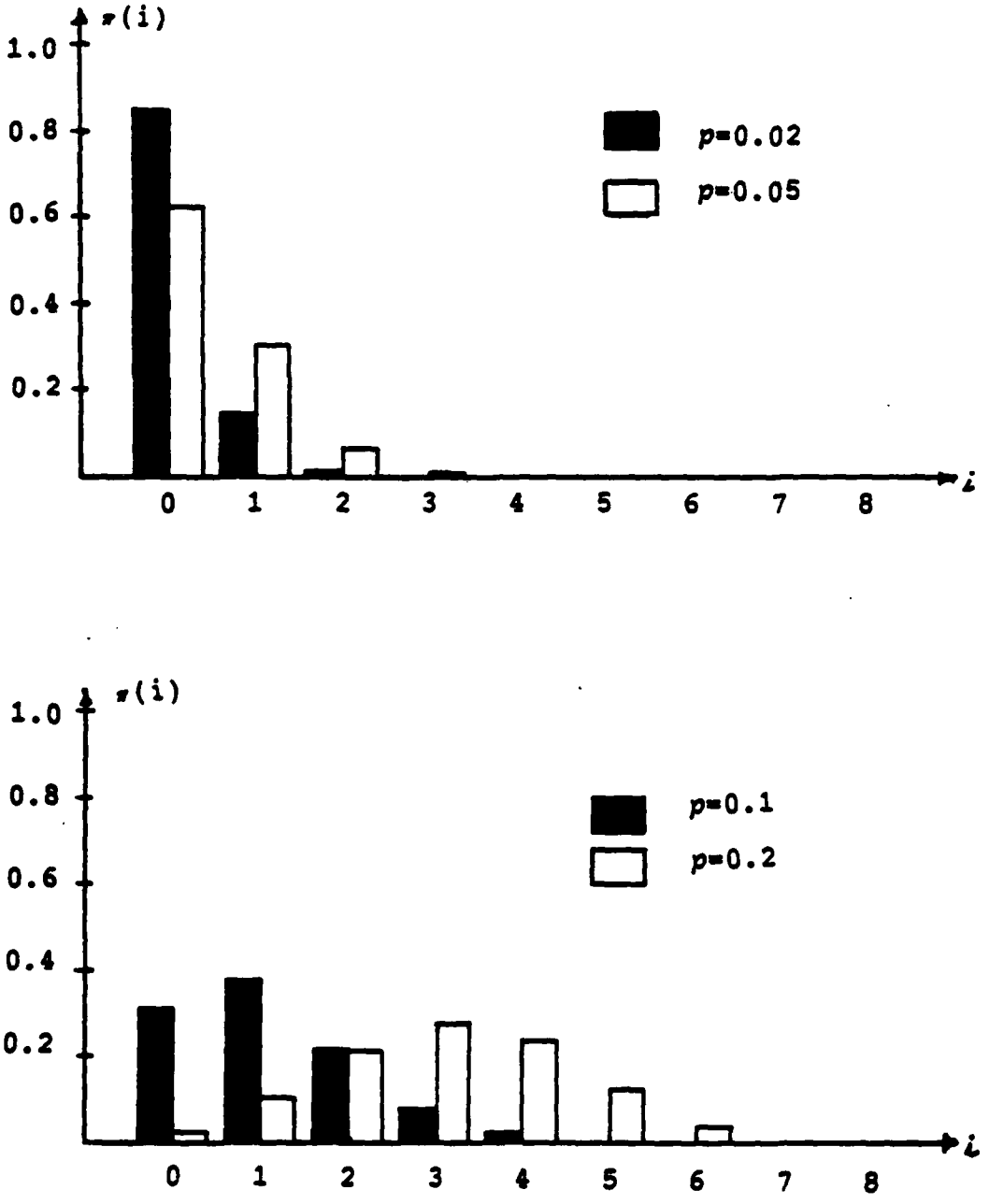


Figure 3.2. Distribution of number of customers in the system, for various p

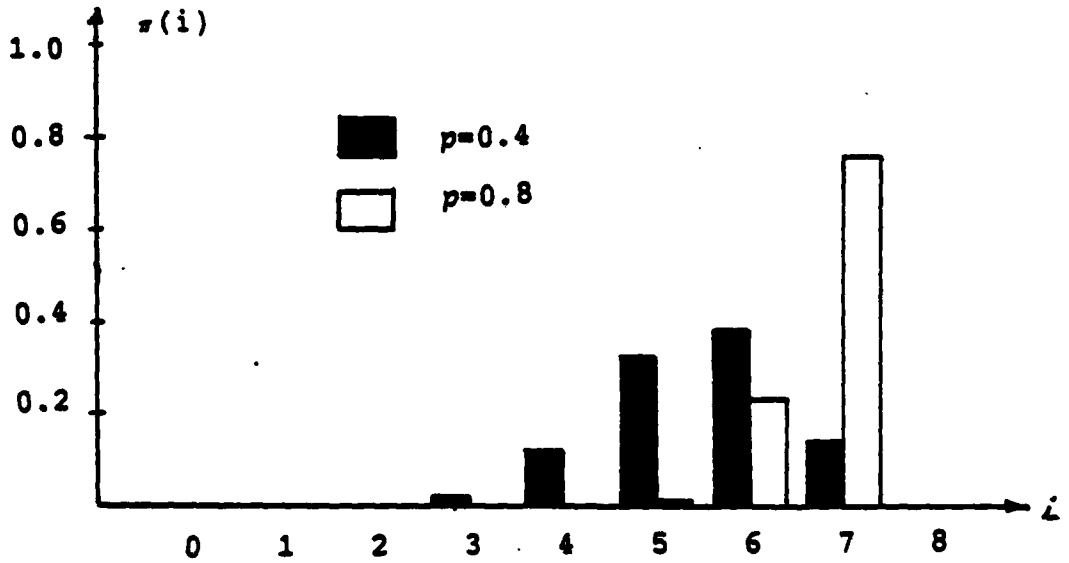


Figure 3.2 (cont'd). Distribution of number of customers in the system, for various p

3.3. Distribution of waiting times in FCFS model

3.3.1. Derivation based on "counting" arguments

$w(i)$ = Pr (waiting time for an arriving customer = i)

We claim that this is equal to

$$w(i) = \frac{\lim_{n\to\infty} 1/n \text{ (\# of arrivals in time n that waited for i)}}{\lim_{n\to\infty} 1/n \text{ (total number of arrivals in time n)}} \qquad (3.1)$$

By the law of large numbers, both limits in the above expression exist and, therefore,

$$w(i) = \lim_{n\to\infty} \frac{\text{\# of arrivals in time n that waited for i}}{\text{total \# of arrivals in time n}}$$

which is the time-averaged probability that the waiting time of an arriving customer is i.

Let r(g,s) = Number of slots up to time n in which g new customers arrive, and s customers are already in the system.

Then, for i = 1,...,n

Number of arrivals in time n that waited i

$$\begin{aligned} &= r(i,0) + r(i+1,0) + \ldots + r(N,0) \\ &\quad + r(i,1) + r(i+1,1) + \ldots + r(N-1,1) \\ &\quad + r(i-1,2) + r(i,2) + \ldots + r(N-2,2) \\ &\quad + \quad \ldots \quad \ldots \quad \ldots \\ &\quad + r(1,i) + r(2,i) + \quad + r(N-i,i) \end{aligned}$$

$$= \sum_{g=i}^{N} r(g,0) + \sum_{s=1}^{i} \sum_{g=i-s+1}^{N-s} r(g,s)$$

$$= r(i,0) + \sum_{s=0}^{i} \sum_{g=i-s+1}^{N-s} r(g,s)$$

This is based on the fact that for every arrival of a block of customers that cause the number of customers in the system to equal or exceed i, exactly one of these customers has a waiting time equal to i.

Now, as $n \to \infty$,

$(1/n)\ r(g,s)$

= Pr (arrival of block of size g, with s customers in system)

= Pr (arrival of block of size g | s customers in the system) x Pr (s customers in the system)

$$= \binom{N-s}{g} p^g (1-p)^{N-s-g}\ \pi(s) \qquad \text{(for } s = 0,\ldots,N,\ g=1,\ldots,N\text{)}$$

Therefore, the numerator of Equation 3.1 is

$$\pi(0) \binom{N}{i} p^i (1-p)^{N-i} + \sum_{s=0}^{i} \pi(s) \left[\sum_{g=i-s+1}^{N-s} \binom{N-s}{g} p^g (1-p)^{N-s-g} \right]$$

Also, as $n \to \infty$, the denominator of Equation 3.1 is

1/n (Total # of arrivals in time n) = E (# of arrivals in a slot)

$$= \sum_{j=0}^{N} \pi(j)\ E(\#\text{ of arrivals in a slot} \mid j \text{ customers already in system})$$

$$= \sum_{j=0}^{N} \pi(j)\ (N-j)\ p$$

which gives, finally,

$$w(i) = \frac{\pi(0)\binom{N}{i}p^i(1-p)^{N-i} + \sum_{s=0}^{i}\pi(s)\left[\sum_{g=i-s+1}^{N-s}\binom{N-s}{g}p^g(1-p)^{N-s-g}\right]}{p\sum_{j=0}^{N-1}\pi(j)(N-j)}$$

$$(\text{for } i = 1,\ldots,N) \qquad (3.2)$$

as the distribution of waiting times in the FCFS model.

This equation was also evaluated for various values of p by the program in Appendix B.1. The waiting time distribution is shown in Figure 3.4, along with those for the round-robin model obtained by simulation.

3.3.2. An alternate derivation

When Equation 3.2 was evaluated, it was observed that for $i = 1,\ldots,N$, $\pi(i)$ and $w(i)$ were proportional; in fact

$$w(i) = \frac{\pi(i)}{1 - \pi(0)} \qquad (\text{for } i = 1,\ldots,N) \qquad (3.3)$$

This is, of course, considerably simpler than Equation 3.2, and I will now proceed to justify this, with the aid of Figure 3.3.

Figure 3.3 shows one busy period. Each dot represents the arrival of a customer with a waiting time (in the FCFS model) represented by the height of the dot. For every dot in the above diagram, there is exactly one time slot for which the number in the system equals the height of the dot. This can be located by moving directly to the right from the

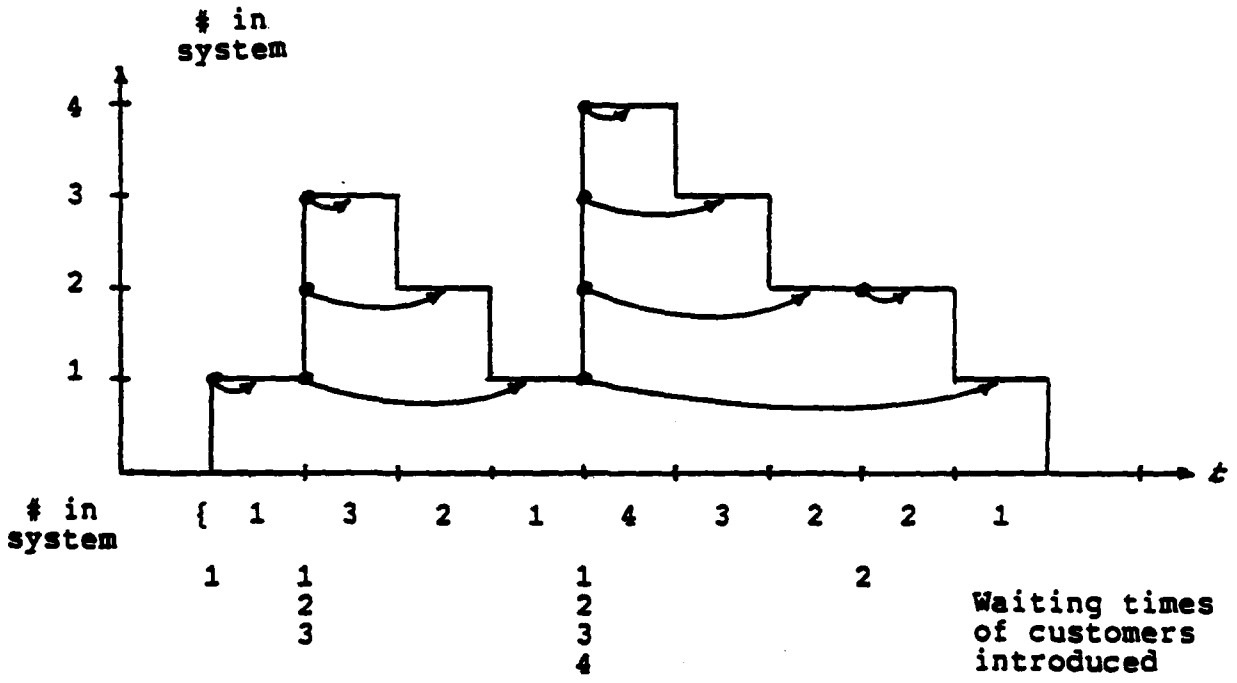


Figure 3.3. One-to-one correspondence between waiting times of arrivals and number of customers in system

dot until the boundary of the "staircase" function is encountered. This correspondence is indicated in Figure 3.3 by arrows.

Another way of looking at this is to visualize the waiting times of arriving customers (assuming a FCFS discipline) to be pushed onto a stack. At each time slot, the waiting time of one customer is popped off the stack. This number corresponds exactly with the number of customers in the system at that time. (Note: This stacking operation

is just a mental convenience; it does not refer to the stacking of customers themselves, which would imply a last-in, first-out discipline.) Thus, in Figure 3.3, the first arrival pushes a 1, which is popped off at the next cycle. The next block of 3 arrivals pushes a 1, a 2 and a 3, which are popped off in reverse order in the next 3 cycles, and so on. Note that 1, 3, 2, 1, ... are just the number of customers in the system in the first 4 time slots of the busy period.

As this holds during busy periods only, we have

# of customers with waiting time i

= # of times the system has i customers in it during a busy period

$$\Rightarrow \Pr(\text{waiting time} = i) = \Pr(\#\text{ in system} = i \mid \text{not idle}) = \frac{\Pr(\#\text{ in system} = i)}{\Pr(\text{not idle})}$$

$$\Rightarrow \quad w(i) = \frac{\pi(i)}{1 - \pi(0)} \qquad (3.3)$$

This result is in fact more general than Equation 3.2, which applies only to the present system. Equation 3.3 applies to any FCFS discrete-time system with a deterministic work-conserving server with unit service rate.

3.4. Simulation results: Waiting time in round-robin model

The simulator, whose listing appears in Appendix B.2, was run for various values of p, to obtain the distributions for the number of customers in the system and the waiting times. As expected, the distribution for $\underline{x}$ was the same as for the FCFS model, which served as a useful check on the simulator, but which was otherwise not too enlightening.

The waiting time distribution was more interesting; again, as expected, the averages were the same for the two models, but the variance was larger for the round-robin

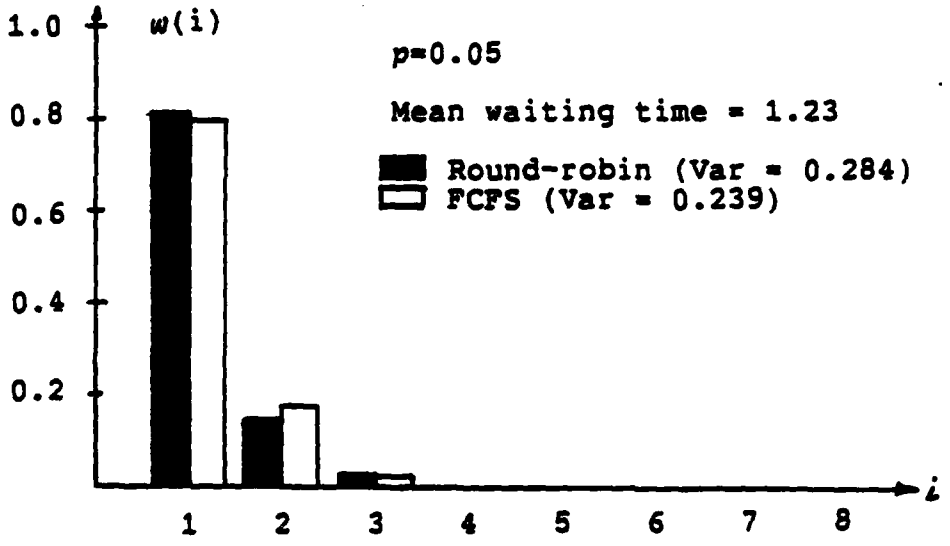
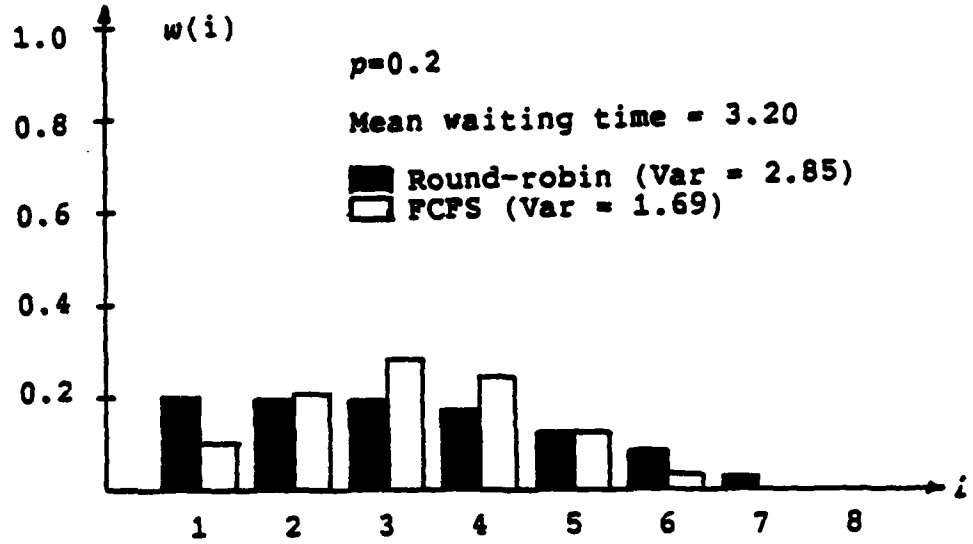


Figure 3.4. Waiting time distribution for round-robin and FCFS models



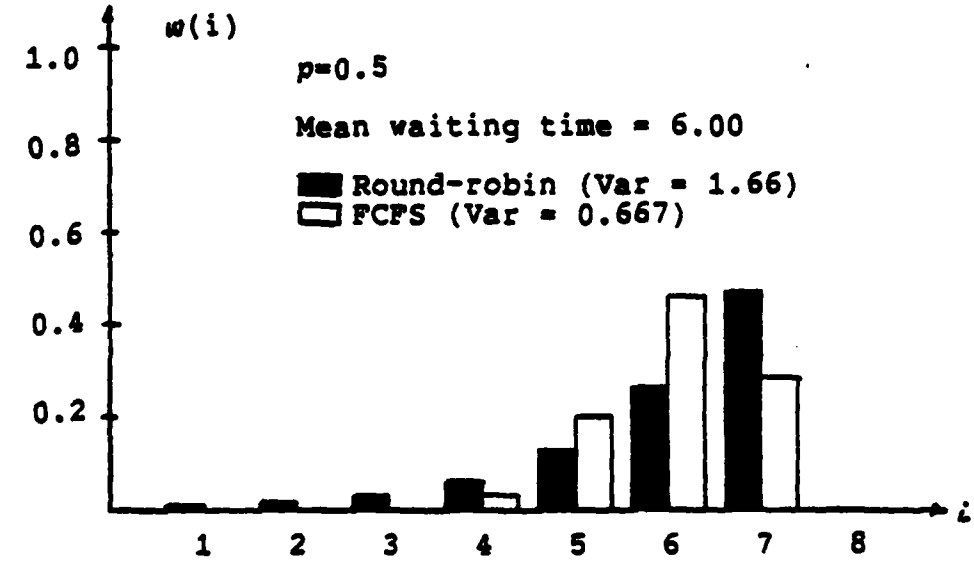


Figure 3.4 (cont'd). Waiting time distribution for round-robin and FCFS models

model, which has a slightly "flatter" distribution. These comparisons are shown in Figure 3.4.

3.5. Idle and busy periods

A parameter that is often of interest in queueing systems is the distribution of the lengths of idle and busy periods. An idle period consists of an interval during which there are no customers in the system, and is preceded and terminated by a departure and an arrival respectively. A busy period consists of an interval during which there is at least one customer in the system, and is preceded and terminated by idle periods.

It is relatively straightforward to obtain an expression for the idle time distribution, and this is done below. The distribution of busy times, however, is a much more complex entity and is resistant to analytic solution. Furthermore, due to the long "tails" that busy time distributions habitually have, it is impractical to obtain this distribution via simulation as extremely long simulations would be required for accurate results. We shall, therefore, be content with obtaining an expression for just the mean length of busy periods.

3.5.1. Distribution of idle periods

For the basic symmetric model, it should be clear that the idle-time (and busy-time) distributions are the same for the FCFS and round-robin systems since, as explained in Section 3.1.3, the number of customers in the system and the

arrival rates do not depend on the serving discipline.

Looking at the FCFS system (Figure 3.1), then, we see that an idle period begins with the system in state 1, followed by a transition to state 0, followed by zero or more self-transitions to state 0 and ends with a transition out of state 0. Therefore, conditioned on the fact that an idle period has just started,

$$I(i) = \Pr(\text{idle time} = i)$$

$$= P(0,0)^{i-1}\,[1 - P(0,0)], \quad (\text{for } i \geq 1) \qquad (3.4)$$

where $P(0,0) = (1-p)^N$ is the probability of a self-transition from state 0 to state 0 (N=number of devices in the system).

This is a simple geometric distribution with mean and variance given by

$$\bar{I} = \frac{1}{1 - P(0,0)}, \qquad \mathrm{Var}(I) = \frac{P(0,0)}{[1 - P(0,0)]^2}$$

3.5.2. Mean length of a busy period

If $\pi(0)$ is the probability that the system is idle (in state 0), then

$$1 - \pi(0) = \text{fraction of time server is busy (see [KLE75])}$$

$$= \lim_{n\to\infty} \frac{\#\text{ of busy cycles}}{\#\text{ of idle cycles} + \#\text{ of busy cycles}}$$

$$= \lim_{n\to\infty} \frac{\bar{B}\,N_B}{\bar{I}\,N_I + \bar{B}\,N_B}$$

where $\bar{B}$ and $\bar{I}$ are the mean busy and idle times respectively,

and N_B and N_I are the number of busy and idle periods respectively. As $n \to \infty$, N_B/N_I approaches unity; therefore

$$1 - \pi(0) = \frac{\overline{B}}{\overline{I} + \overline{B}}$$

$$\Rightarrow \overline{B} = \overline{I}\,[1/\pi(0) - 1] \qquad (3.5)$$

where $\overline{I}$ is as given in Section 3.5.1 above.

3.6. Fairness characteristics of the biased arbiter

Recall from Section 2.3.1 that, when originally designed, the arbiter had an inherent source of unfairness. This was caused by the fact that when two or more devices simultaneously make requests after the arbiter has been idle, the lower-numbered one will always "win." For reasons explained in that section, it was not believed that this unfairness would be significant, a belief that was refuted by simulations which showed that under certain load conditions, lower numbered devices were significantly better off. To make matters worse, the request rate at which this effect is most pronounced is in the neighbourhood of the request rate at which the CLUMPS system is expected to operate.

3.6.1. Simulation results

The results of these simulations are shown in Table 3.1 below. The parameter used to measure the unfairness is the difference in mean waiting times for the lowest and highest numbered devices. This parameter is readily obtained from

Prob. (p)	Mean waiting times dev 0	dev 7	overall	Δw	I_o
.01	1.003	1.063	1.034	5.9 %	.071
.02	1.018	1.142	1.073	11.6 %	.126
.05	1.113	1.376	1.229	21.4 %	.210
.07	1.237	1.536	1.365	21.9 %	.215
.09	1.410	1.740	1.537	21.5 %	.195
.10	1.502	1.832	1.643	20.1 %	.178
.15	2.249	2.396	2.324	6.3 %	.080
.20	3.179	3.264	3.194	2.7 %	.021

Table 3.1. Smallest and largest mean waiting times for devices, at various request probabilities

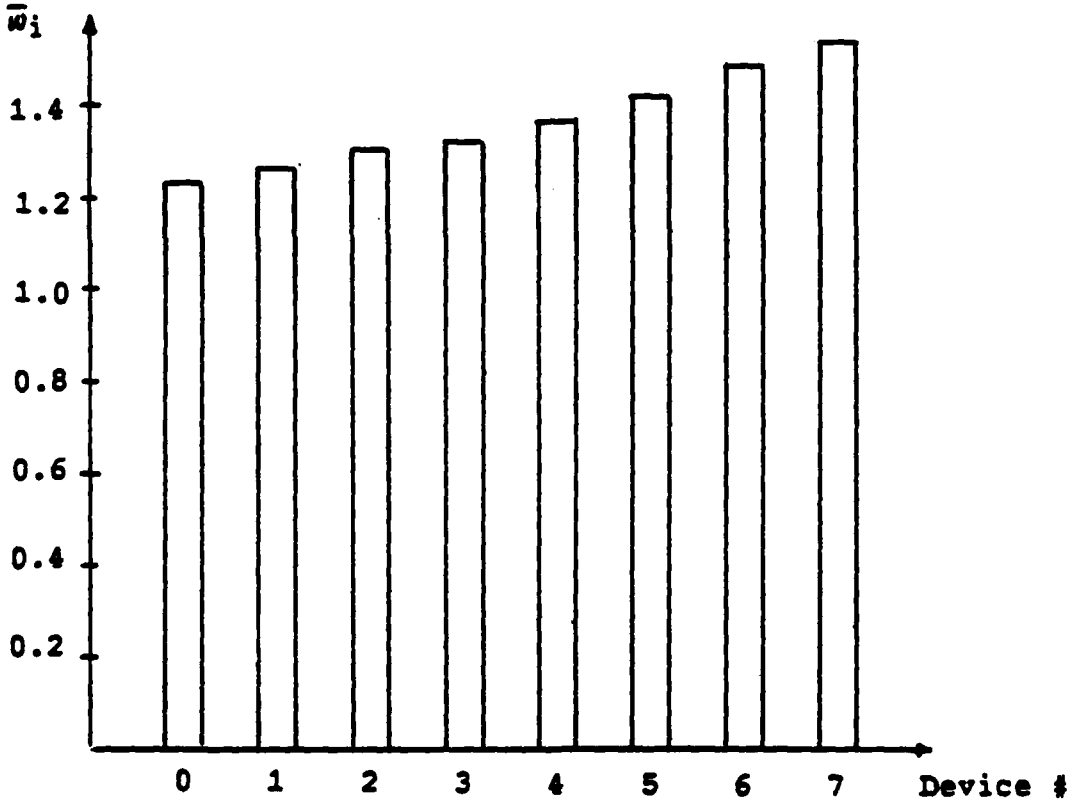


Figure 3.5. Mean waiting time for all devices using biased arbiter (p=0.07)

the simulator output (cf. Appendix B.2). On the basis of these simulations, it was decided to modify the arbiter design, as explained in Section 2.3.1, to remove this source of unfairness.

Table 3.1 summarizes the results for various values of p up to p=0.2 (beyond that, the system was not idle often enough during the simulation to get meaningful results for this measurement). For each value of p, we show the mean waiting time for device 0 (the most favoured) and device 7 (the least favoured). The overall average waiting time for all devices is also shown, as is Δw, the percent difference between devices 0 and 7 with respect to this average.

Clearly, for values of p in the neighbourhood of 0.07, the difference can be fairly significant (about 20%), revealing that under certain load conditions, lower numbered devices can have significant advantage. Figure 3.5 shows the mean waiting time for all 8 devices for p=0.07; the increase in mean waiting time with increasing device number seems to be rather steady.

The device waiting times for the fair arbiter (cf. Section 2.3.1) were all the same and, of course, equal to the overall average waiting time.(1)

---

(1) The overall average waiting time is the same for the fair and biased arbiters since the server is

3.6.2. Relation to frequency of idle periods

Since the source of unfairness is the idle periods, it seems reasonable that the difference in mean waiting time between devices 0 and 7 should be directly related to the frequency of idle periods. To verify this hypothesis, we compute the mean number of idle periods per cycle, I_0, as follows:

Let $I_0 = \lim_{n\to\infty} 1/n$ [number of idle periods in n cycles]

$= \lim_{n\to\infty} 1/n$ [(# of idle periods in n cycles) x $\bar{I}$]

(where $\bar{I}$ is the mean length of an idle cycle)

$= 1/\bar{I} \lim_{n\to\infty} 1/n$ [# of idle cycles in n cycles]

$= \pi(0)/\bar{I}$

Using the expression for $\bar{I}$ from Section 3.5.1, this is:

$$I_0 = \pi(0)\,[1 - (1-p)^8] \qquad (3.6)$$

This is also tabulated in Table 3.1, and one observes a striking correspondence between I_0 and Δw.

In all results presented from this point onward, the biased round-robin arbiter shall not be considered and all references to the round-robin arbiter shall be to the fair one.

---

work-conserving.

4. EFFECT OF SINGLE HEAVY OR LIGHT USER ON FAIRNESS

4.1. Round-robin arbiter

As in Section 3, we continue to use the fair arbiter (cf. Section 2.3.1) in the simulations of this section. We also continue to use the basic symmetric model of Section 3, with the exception that one device (which, without loss of generality, we can let be device #0) has a request probability different from the others (devices #1-#7).

Let p now refer to the request probability in an idle cycle of any device numbered 1-7 and let p_0 be the request probability, in an idle cycle, of device number 0. In this section, we shall examine the effect of this imbalance on fairness, as measured by the mean waiting time of each device.(1)

The results of these simulations are presented in Table 4.1. In summary, the following observations were made:

1) For all combinations of p and p_0 values that were

---

(1) This type of performance analysis is often done to ensure that a single heavy user does not unfairly pre-empt other users. Such a situation (of a heavy user amongst many light users) will, in fact, occur as a matter of course in the CLUMPS system; as mentioned in Section 1.3, memory is used either to store pseudo-opcodes or to store data. Access to the pseudo-opcodes is regular though relatively infrequent as a processor has to interpret and then execute the opcode. This could take several tens of cycles during which time no requests will be generated by that processor. Data transfers, on the other hand, are irregular but much more bursty and could need several hundred consecutive memory cycles.

simulated, the fact that the request probability for device 0 differed from that of devices 1-7 resulted, surprisingly enough, in no measurable skew in the waiting times for devices 1-7. In other words, there were no measurable differences in the waiting times for devices 1-7 caused by values of p_0 that differed substantially from p. This observation is used to advantage in simplifying Table 4.1; the mean waiting times of devices 1-7 are lumped together as one parameter since they are essentially equal. In Table 4.1, w_0 refers to the mean waiting time for device #0 and w to that for devices 1-7. w' is the overall mean waiting time and Δw is the percentage difference between w_0 and w with respect to this overall mean.

2) At <u>light loading</u> ($p \simeq 0.01$-0.05, $w \simeq 1$-1.5), there was no significant difference between the waiting time for device 0 and that for the other devices, for all values of p_0 between 0.01 and 1.0. <u>Note</u>: This is the load range at which the system is expected to run most of the time, during execution of pseudo-opcodes.

3) At <u>medium loads</u> ($p \simeq 0.1$-0.2, $w \simeq 1.5$-4), device #0 had a slightly longer waiting time for values of p_0 that differed substantially from p, i.e., device #0 waited longer when its request probability was very small <u>or</u> very big compared to that of devices 1-7. For $p_0 >> p$, the difference was more pronounced.

4) At <u>heavy loads</u> ($p \geq 0.5$, $w \simeq 4\text{-}7$), the system was constantly busy during each simulation of 100,000 cycles. Under these heavily loaded conditions, the device with the smaller request probability had the shorter waiting time, i.e., for $p_o < p$, w_o was less than w and for $p_o > p$, w_o was greater than w.

p_o	w_o	w	w'	Δw
0.001	1.04	1.03	1.03	+0.5 %
0.02	1.04	1.04	1.04	0 %
0.1	1.06	1.05	1.05	+ 1 %
0.4	1.07	1.07	1.07	0 %
0.8	1.07	1.06	1.07	+ 1 %

Table 4.1 (a): $p = 0.01$

p_o	w_o	w	w'	Δw
0.01	1.22	1.19	1.19	+2.5 %
0.1	1.24	1.26	1.25	-1.5 %
0.8	1.40	1.40	1.40	0 %

Table 4.1 (b): $p = 0.05$

p_o	w_o	w	w'	Δw
0.02	1.56	1.51	1.51	+ 3 %
0.2	1.73	1.75	1.75	- 1 %
0.8	2.07	1.98	2.02	+ 5 %

Table 4.1 (c): $p = 0.1$

p_o	w_o	w	w'	Δw
0.01	2.60	2.55	2.55	+ 2 %
0.1	2.94	2.94	2.94	0 %
0.5	3.87	3.50	3.57	+10 %
0.8	4.38	3.57	3.72	+22 %

Table 4.1 (d): $p = 0.2$

p_0	w_0	w	w'	Δw
0.01	4.02	5.07	5.06	-21 %
0.1	4.37	5.54	5.45	-21 %
0.8	6.69	6.00	6.09	+11 %

Table 4.1 (e): $p = 0.5$

p_0	w_0	w	w'	Δw
0.1	4.46	6.27	6.15	-29 %
0.5	6.06	6.74	6.66	-10 %
1.0	7.00	6.75	6.78	+ 4 %

Table 4.1 (f): $p = 0.8$

Table 4.1. Waiting times for device 0 and other devices as a function of request probabilities of device 0 and other devices (round-robin arbiter).

4.2. First-come, First-served (FCFS) arbiter

A common problem with many FCFS systems is that a heavy user can tie up an unfair share of the system resources. Although the CLUMPS system does not use a FCFS arbiter, it is interesting to see if this is still a problem in the case where requests can be made only by inactive devices, i.e., the queue capacity for each device is unity.

Simulations similar to those in Section 4.1 were run, except that a FCFS arbiter was used instead of a round-robin one. Table 4.2 shows the results of these simulations. As in the round-robin case, the waiting times for devices 1-7 were not measurably different, so they will collectively be referred to as w while that for device 0 will be called w_0. The results are summarized below:

1) At low to medium request probabilities ($p \leq 0.2$, $w \leq 4$),

Table 4.2 shows that the heavier user has a shorter mean waiting time, i.e., for $p_o<p$, w_o was greater than w and for $p_o>p$, w_o was less than w.

2) At high request probabilities ($p\geq 0.5$, $w>4$), device #0 (the non-uniform one) seemed to have a longer mean waiting time for $p_o>p$ as well as for $p_o<<p$. However, for p_o less than, but in the neighbourhood of p, w_o was slightly less than w.

p_o	w_o	w	w'	Δw
0.005	1.03	1.03	1.03	0 %
0.02	1.03	1.04	1.04	-0.5 %
0.1	1.04	1.08	1.06	- 4 %
0.5	1.04	1.21	1.07	-16 %

Table 4.2 (a): $p = 0.01$

p_o	w_o	w	w'	Δw
0.01	1.22	1.19	1.19	+ 2 %
0.1	1.24	1.27	1.26	- 2 %
0.5	1.26	1.49	1.38	-17 %

Table 4.2 (b): $p = 0.05$

p_o	w_o	w	w'	Δw
0.02	1.60	1.51	1.51	+ 6 %
0.2	1.68	1.77	1.75	- 5 %
0.6	1.80	2.10	2.00	-15 %

Table 4.2 (c): $p = 0.1$

p_o	w_o	w	w'	Δw
0.01	2.84	2.53	2.53	+12 %
0.1	3.04	2.90	2.91	+ 5 %
0.5	3.48	3.65	3.62	- 5 %
0.8	3.68	3.86	3.83	- 5 %

Table 4.2 (d): $p = 0.2$

p_o	w_o	w	w'	Δw
0.02	5.55	5.13	5.13	+ 8 %
0.1	5.65	5.47	5.48	+ 3 %
0.4	5.93	5.94	5.94	0 %
0.8	6.17	6.08	6.09	+ 2 %

Table 4.2 (e): $p = 0.5$

p_o	w_o	w	w'	Δw
0.05	6.31	6.03	6.04	+ 5 %
0.2	6.40	6.42	6.42	-0.5 %
0.6	6.63	6.71	6.70	- 1 %
1.0	6.86	6.77	6.78	+ 1 %

Table 4.2 (f): $p = 0.8$

Table 4.2. Waiting times for device 0 and other devices as a function of request probabilities of device 0 and other devices (FCFS arbiter).

4.3. Round-robin vs. FCFS

Sections 4.1 and 4.2 show some of the characteristics of the round-robin and FCFS arbiters under request rates that are asymmetric with respect to different devices. These results are consolidated and summarized below:

1) If waiting time is used as a measure of fairness, and if it is considered desirable that all devices have the same mean waiting time regardless of their request rates, then one can conclude that the round-robin arbiter is more

fair than the FCFS arbiter. This is because the latter consistently shows a difference between w_o and w whenever p_o and p differ, while the former retains equality between w_o and w for p up to about 0.1 and only begins to show a difference when the load starts to become heavy ($p \geq 0.1$).

2) Where a difference exists between w_o and w for the round-robin arbiter (i.e., for $p \geq 0.1$), it seems to penalize the heavy users by giving them slightly longer waiting times. The FCFS arbiter however, at least for p up to about 0.2, seems to give preference to heavy users by giving them somewhat shorter waiting times.

5. EFFECT OF LONGER GUARANTEED IDLE TIMES AFTER REQUESTS

5.1. Relaxation of restrictions on present model

In all analyses and simulations done so far, it has been assumed that every idle device has some constant probability (possibly different for different devices) of generating a request during any cycle it is idle, except that it cannot generate a request in the cycle immediately following the one in which it was served. In other words every device has a guaranteed idle time of at least one cycle between requests. As explained in Section 3, this model was chosen as it is simple to understand and analyse and is a reasonable model of the actual hardware.

The simulator program (cf. Appendix B.2) is capable of handling more complex request rate structures; specifically, for each device, the probability of a request in any cycle can be an arbitrary function of the device number as well as the number of cycles that that device has been idle. In this section, this feature of the simulator shall be utilized in examining the effect of increasing the number of guaranteed idle cycles between successive requests (so far, this has just been one cycle).(1)

---

(1) This enhancement also models the behaviour of the CLUMPS system more closely. Recall from the footnote in Section 4.1 that when the processors are accessing pseudo-opcodes from global memory (which they will be doing most of the time), a memory read is followed by several tens of cycles during which time the processor interprets the pseudo-opcode and will usually not access

5.2. Decreasing randomness

For any device, let m be the number of guaranteed idle cycles between the end of a service for that device and the earliest time that another request could be initiated by the same device. We let m be any positive integer greater than or equal to zero.(1) For simplicity, we shall assume that after the first m idle cycles, the device has a constant probability of request p for any subsequent idle cycle. Assume, furthermore, that m and p are the same for all devices.

To study the effect of varying m without changing the effective request rate, we shall fix the mean time between the end of a service for a device and the generation of a request for the same device (excluding service times). For fixed p, m, this mean time to request, which we shall call T, is given by:

$$
\begin{aligned}
T &= mp + (m+1)p(1-p) + (m+2)p(1-p)^2 + \ldots \\
&= mp\,[1 + (1-p) + (1-p)^2 + \ldots] \\
&\quad + p(1-p)\,[1 + 2(1-p) + 3(1-p)^2 + \ldots] \\
&= m + 1/p - 1
\end{aligned}
$$

---

global memory.

(1) m=0 cannot be physically realized in the CLUMPS system as explained in Section 3.1; it may, however, be useful to compare the results for other values of m with the m=0 case.

For a fixed T, then, p is a function of m; we shall denote this by subscripting it with the letter m. Thus,

$$p_m = [T - m + 1]^{-1} \qquad (5.1)$$

For consistency with earlier notation, we shall denote p_1 by p. It is clear, then, that the mean time to request is given simply by

$$T = 1/p \qquad (5.2)$$

From equation 5.1, we can see that as m increases, so does p_m and, at the maximum value of m=T (assuming T is an integer), p_m=1. Thus, increasing m can be viewed as decreasing the randomness of the system or increasing its deterministic behaviour. In fact, at m=T and p_m=1, the system is totally deterministic; each device generates a request with probability 1 after exactly T idle cycles.

Figure 5.1 shows the mean waiting time as a function of m, for various values of T. Intuitively, one would expect that performance would improve (the mean waiting time would decrease) as the system becomes more deterministic (i.e., as m gets closer to T). In the extreme case of m=T, p_m=1, it is easy to see that, as long as T≥7, the requests will automatically space themselves out so that there are, in the steady state, never any simultaneous requests. In this case, the total waiting time of a customer would just be the 1 unit of service time, which is the best performance possible. One might expect that there would be a steady improvement in performance as m increased from m=0

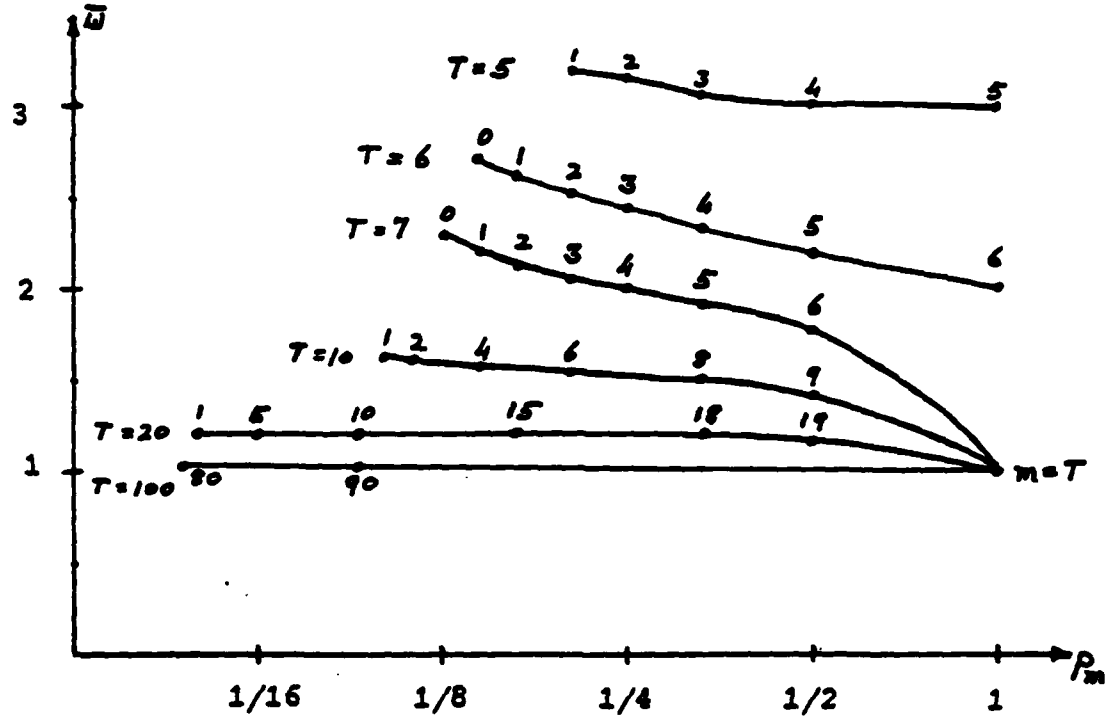


Figure 5.1. Mean waiting time as a function of number of guaranteed idle cycles, for various values of "mean time to request."

($p_m = [T+1]^{-1}$) to $m=T$ ($p_m=1$). Figure 5.1 shows that this is indeed the case. For convenience, Figure 5.1 has been scaled so that it actually appears to be a function of p_m. The value of m is shown next to every sample point in the figure.

6. CONCLUSIONS AND SUMMARY

We have studied the performance characteristics of the round-robin arbiter used in the CLUMPS multi-processor system and have compared some of the characteristics with those of a First-come, First-served (FCFS) arbiter. A round-robin arbiter was chosen for implementation due to its simplicity and better fairness qualities in some situations.

For this study, we used a simplified "quasi-memoryless" model of arrival statistics. The waiting time distribution of the round-robin arbiter was found to have the same mean but larger variance than that of a FCFS arbiter. The idle time distribution for the arbiter was found to have a simple geometric distribution.

The effect of a single user on the system fairness was measured by observing the effect of a heavy user on the mean waiting time of each user. Under the type of loading conditions expected during operation of the CLUMPS system, the round-robin arbiter was found to be fairer (with the assumed fairness criterion) than a FCFS arbiter.

Finally, we studied the effect of increasing the deterministic behaviour of requests while keeping the mean time between requests constant. It was found that performance (as measured by mean waiting time) improved uniformly as the system was made more deterministic.

In the course of these analyses, a simple result was obtained for the waiting time distribution of any

discrete-time FCFS system with a single work-conserving deterministic server.

REFERENCES

[BUR82] W.F. Burger, N. Halim et al, "Draft NIL Reference Manual," IBM Research Report 9732 (42993), 8 December, 1982.

[GAL83] Robert G. Gallager, "The Dynamics of Data Network Research," NSF Proposal, February 1983.

[HEL82] Daniel Helman, Mark Tubinis, "CLUMPS Proposal," Unpublished Report, 22 Feb, 1982.

[HEL83] Daniel Helman, "CLUMPS Architecture (model 2)," CLUMPS Project Internal memo, Laboratory for Information and Decision Systems, MIT, Rev: 16 February, 1983.

[HEL84] Daniel Helman, "Packet Radio Simulation and Protocol," Thesis proposal, Laboratory for Information and Decision Systems, MIT, To be submitted in August, 1984.

[HIT80] Hitachi, "IC Memories Data Book," 1980, Hitachi America, San Jose, California.

[INT82] Intel, "Components Data Catalog," January 1982, Intel Corporation, Santa Clara, California.

[KLE75] Leonard Kleinrock, "Queueing Systems, Volume I," John Wiley and Sons, 1975, pp. 17-19.

[LEB84] Stephen J. LeBlanc, "6.100 Lab Report on the CLUMPS Switch," Submitted to Profs. L.A. Gould and R.G. Gallager, MIT, 5 June, 1984.

[MER82] Shahrukh S. Merchant, "Lab notes on CLUMPS Hardware," Unpublished document, September 1982-July 1984.

[MOT81] "MC68A09E 8-bit Microprocessing Unit Data Sheet," Motorola Semiconductors, Austin, Texas, 1981.

[NAT82] National Semiconductors, "PAL Databook," 1982.

[NIK84] Georgios N. Nikolaou, "Table Implementation for the Network Implementation Language (NIL) Compiler," S.B. Thesis, MIT, February 1984.

[PAR82] F.N. Parr and R. Strom, "NIL: A Programming Language for Software Architecture," IBM Research Report 9227 (40550), 25 January, 1982.

[POE79] Elmer C. Poe and James C. Goodwin II, "The S-100 and Other Micro Buses," 2nd edition, 1979, Howard W. Sams & Co., Indianapolis, Indiana.

[TUB83] Mark A. Tubinis, "Implementation of a Data Communications Node Utilizing a Novel Multiprocessing Architecture," B.S. Boston University, 1981, S.M. MIT, Feb 1983.

APPENDIX-A

A. HARDWARE DESCRIPTION

A.1. CLUMPS Global bus specifications

Figure 1.2 gives a conceptual view of a typical CLUMPS node. Figure A.1 shows a node from the implementation

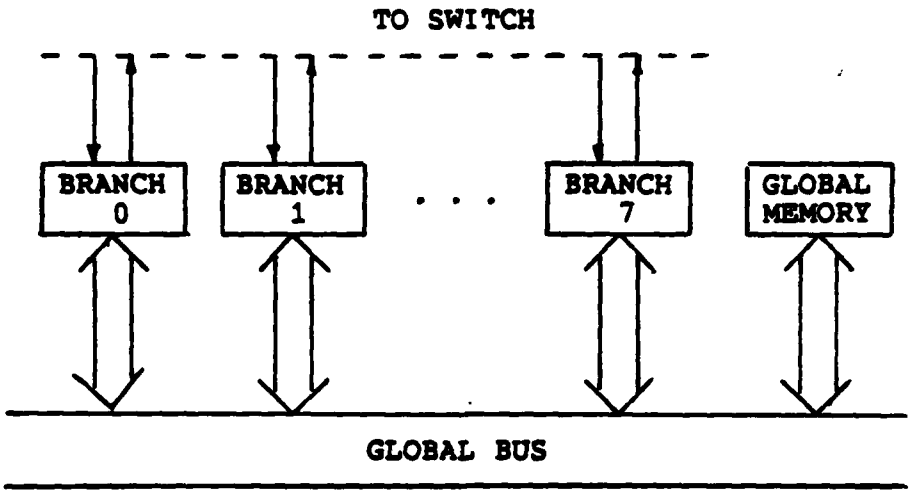


Figure A.1. Implementation of a CLUMPS node.

standpoint. All branches and the global memory share a common bus called the global bus. However, branches can communicate to each other only via global memory.

A.1.1. Bus signals

The global bus is a modified version of the Pro-Log STD

Pin #s	Name	Function
1,2	+5V	Power
3,4	GND	Ground
5,6	-5V	Unused
13,11,9,7, 14,12,10,8	D0- D7	Global data (bidirectional)
29,27,25,23, 21,19,17,15, 30,28,26,24, 22,20,18,16	A0 - - A15	Global address (input)
31,33,35,37, 39,41,43,45	$\overline{ACK0}$- $\overline{ACK7}$	Acknowledgements for branches 0-7 (output)
32,34,36, 38,40,42, 44,46	$\overline{REQ0}$- $\overline{REQ7}$	Requests for branches 0-7 (unused lines should be pulled up to +5V externally) (input)
47	$\overline{RESETOUT}$	Signal to reset node (output)
48	$\overline{NMIOUT}$	Signal to generate non-maskable interrupt to all branch processors (input)
49	CLOCKOUT	Generates TTL level clock output at 2.4576 MHz nominal frequency (1/8 internal clock frequency)
50	R/$\overline{W}$	Read/write (open-collector, input)
51-52	--	Unused
53-54	AUXGND	Auxiliary ground
55	AUX +V	+12 V
56	AUX -V	-12V

Table A.1. CLUMPS Global bus pinouts

bus [POE79]. Table A.1 shows the bus pinout designations and specifies whether the signals are inputs or outputs with respect to the global memory board. Note that several signals differ from the standard STD bus specification.

While the global memory board uses all ACK and REQ signals, note that a branch board would use only one of

Address range (hex)	Use
0000 - 007F	Global RAM
(0080 - 0086 even)	Node ID (read only)
(0081 - 0087 odd)	Branch ID (read only)
(0088, 0089)	8254 Counter #0
(008A, 008B)	8254 Counter #1
(008C, 008D)	8254 Counter #2
(008E, 008F)	8254 Control word register
(0090 - 0097)	NMI (on write)
(0098 - 009F)	RESET (on write)
0090 - FFFF	Global RAM

Table A.2. CLUMPS global memory board

each, corresponding to its branch identification number (0-7).

A.1.2. Global memory map

The memory map for the global memory board is shown in Table A.2. All addresses are in hexadecimal. Parentheses around an address range or address list indicate that all addresses in that range or list are equivalent (this was done primarily to simplify address decoding).

A read from the "Node ID" location allows a branch processor to read the identification number (0-255) of its node. This is set via DIP switches on the global memory board. A read from the "Branch ID" location returns the number (0-7) of the branch which generated the read request.

Locations 0088-008F apply to the Intel 8254 Programmable Interval Timer [INT82] on the global memory board.

A write to any location in the range 0090-0097 will

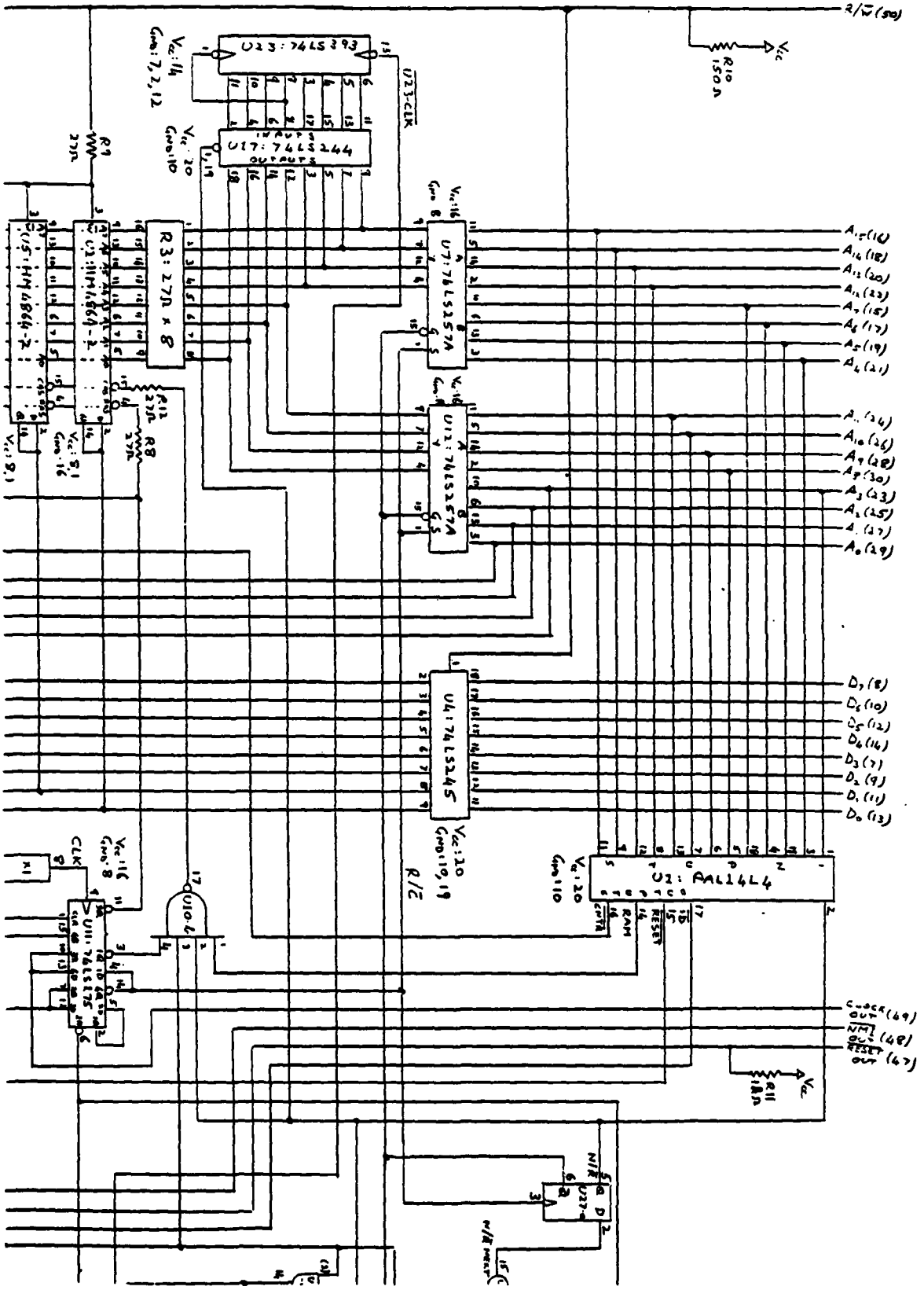
generate a pulse on the NMIOUT line on the bus. This triggers a non-maskable interrupt on all the branch processors. As of this writing, no specific use has been assigned to this feature. A write to any location in the range 0098-009F generates a pulse on the bus RESETOUT line. This causes all peripheral boards on the bus (branch boards in particular) to undergo a reset. The global memory board does not reset itself with this operation.

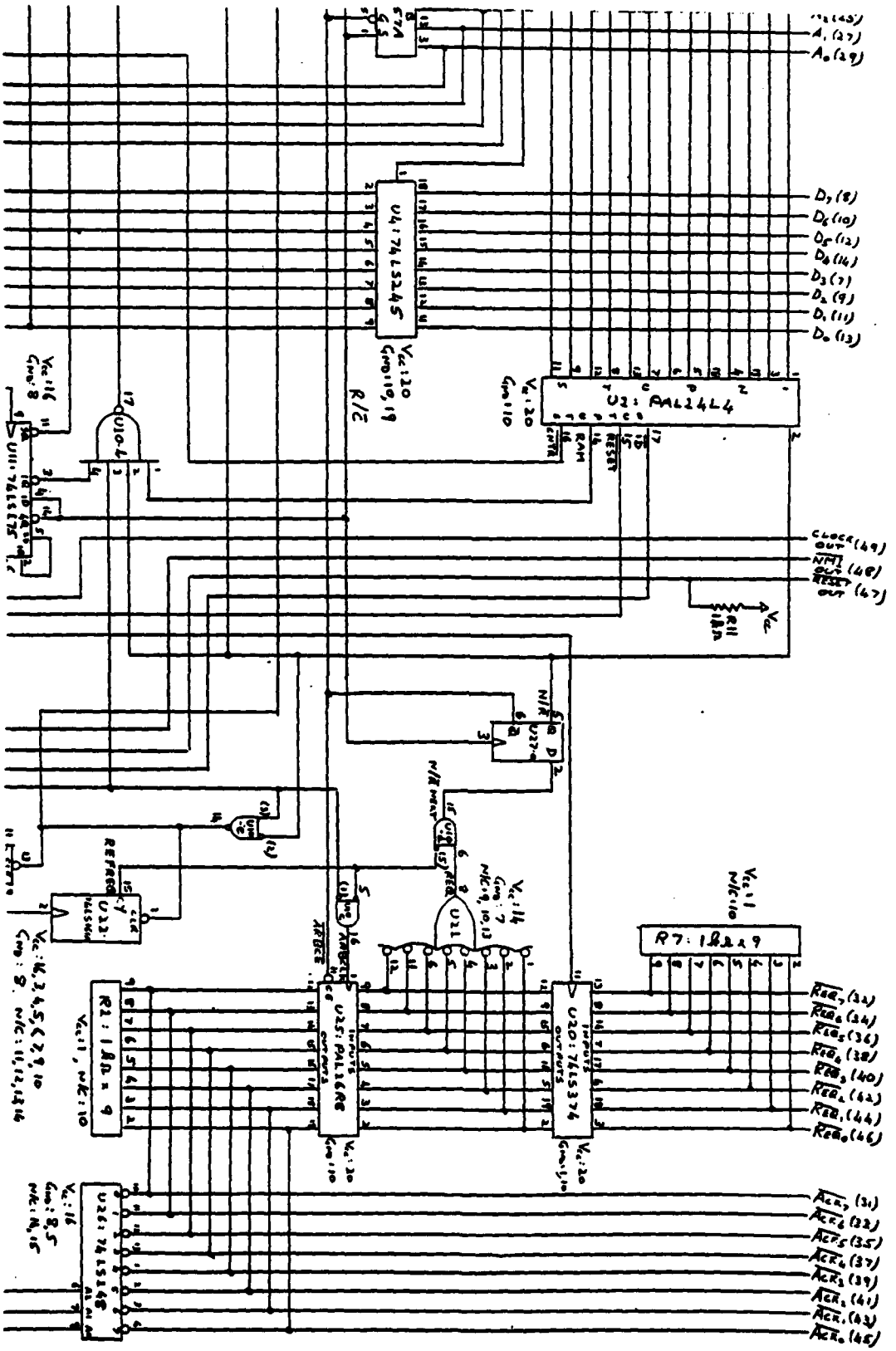
Note that the NMI and RESET locations overlap with global read/write memory. This was done intentionally, so that a processor that generates an NMI or RESET may simultaneously write a word into memory. This could, perhaps, be its branch identification, for it may be desirable to know, after initiation of an NMI or RESET, the source of this NMI or RESET.

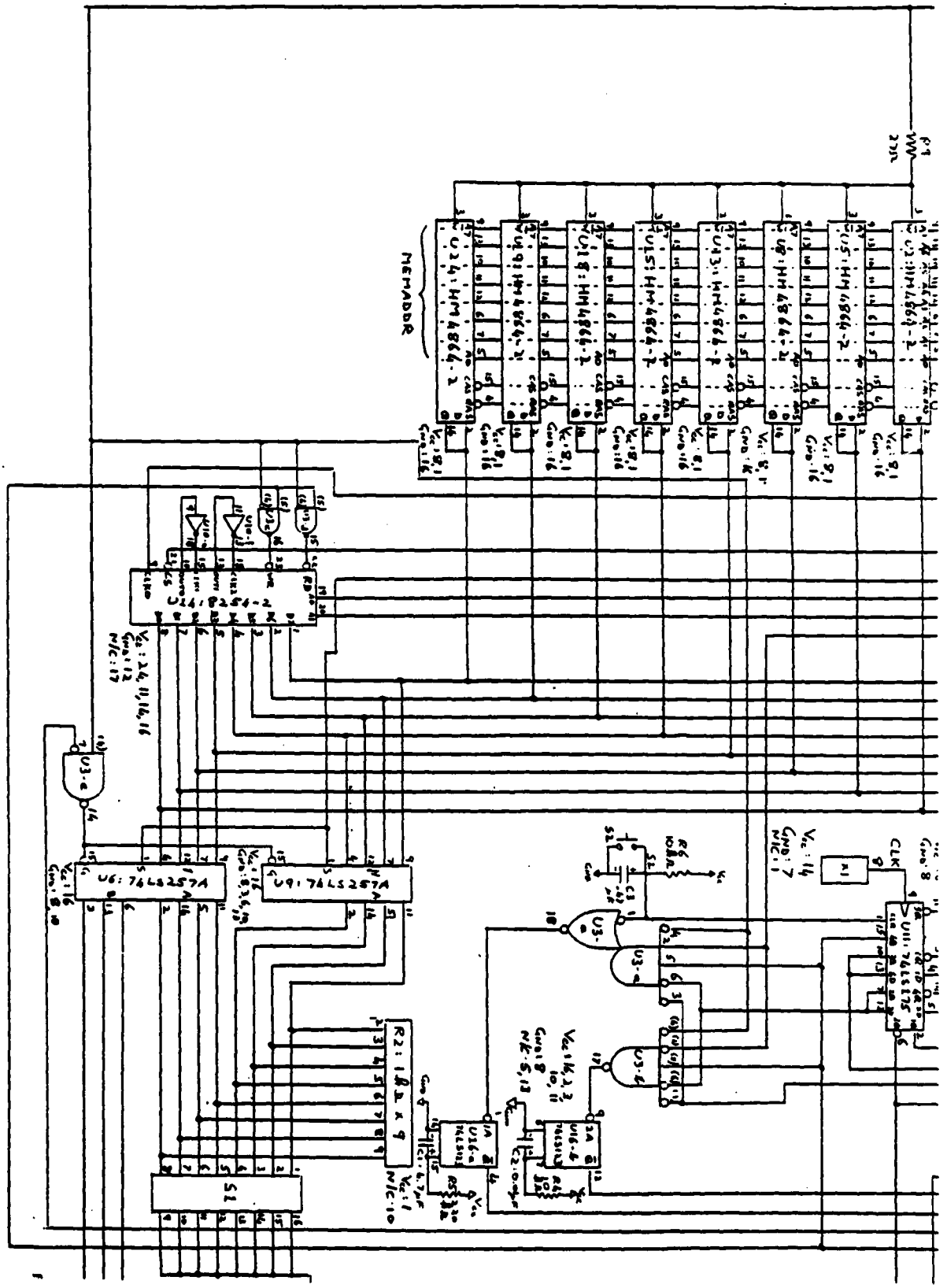
A.2. Global memory board design

A.2.1. Schematic diagram

Figure A.2 on the foldout (and replicated on the four pages following) is a complete circuit diagram of the global memory board, including pin and part identification.


R/W (50)
U23: 74LS393
U17: 74LS244
U7: 74LS524A
U12: 74LS524A
U4: 74LS245
U1: PAL14L4
R3: 27Ω x 8
R8
R9
R10 150Ω
R11 180Ω
U23-CLK
CLOCK OUT (49)
RESET OUT (47)
R/W
A15 (1)
A14 (18)
A13 (20)
A12 (23)
A7 (15)
A6 (17)
A5 (19)
A4 (21)
A3 (24)
A2 (26)
A1 (28)
A0 (29)
D7 (8)
D6 (10)
D5 (12)
D4 (14)
D3 (7)
D2 (9)
D1 (11)
D0 (13)


U4: 74LS245
U2: PAL24L4
U20: 74LS374
U25: PAL16R8
U26: 74LS148
U21
U10-4
R7: 1kΩx9
R1: 1kΩx9
R/W
REFRESH
CLOCK OUT (49)
NMI OUT (48)
RESET OUT (47)
D7 (8)
D6 (10)
D5 (12)
D4 (14)
D3 (7)
D2 (9)
D1 (11)
D0 (13)
A1 (27)
A0 (29)
Reg7 (34)
Reg6 (34)
Reg5 (36)
Reg4 (38)
Reg3 (40)
Reg2 (42)
Reg1 (44)
Reg0 (46)
Acc7 (31)
Acc6 (33)
Acc5 (35)
Acc4 (37)
Acc3 (39)
Acc2 (41)
Acc1 (43)
Acc0 (45)


MEMADDR
U24: HM4864-2
U23: HM4864-2
U22: HM4864-2
U21: HM4864-2
U20: HM4864-2
U19: HM4864-2
U18: HM4864-2
U17: HM4864-2
U14: 8254-2
U6: 74LS257A
U9: 74LS257A
U3-a
U3-b
U3-c
U3-d
U11: 74LS125
U16-a
U16-b
U16-c
U16-d
R2: 1kΩ x 9
51
CLK
Vcc: 14
GND: 7
N/C: 1
Vcc: 24, 11, 10, 15
GND: 12
N/C: 17
Vcc: 16
GND: 8
Vcc: 14, 2, 10, 11
GND: 8
N/C: 5, 13
Vcc: 1
N/C: 10

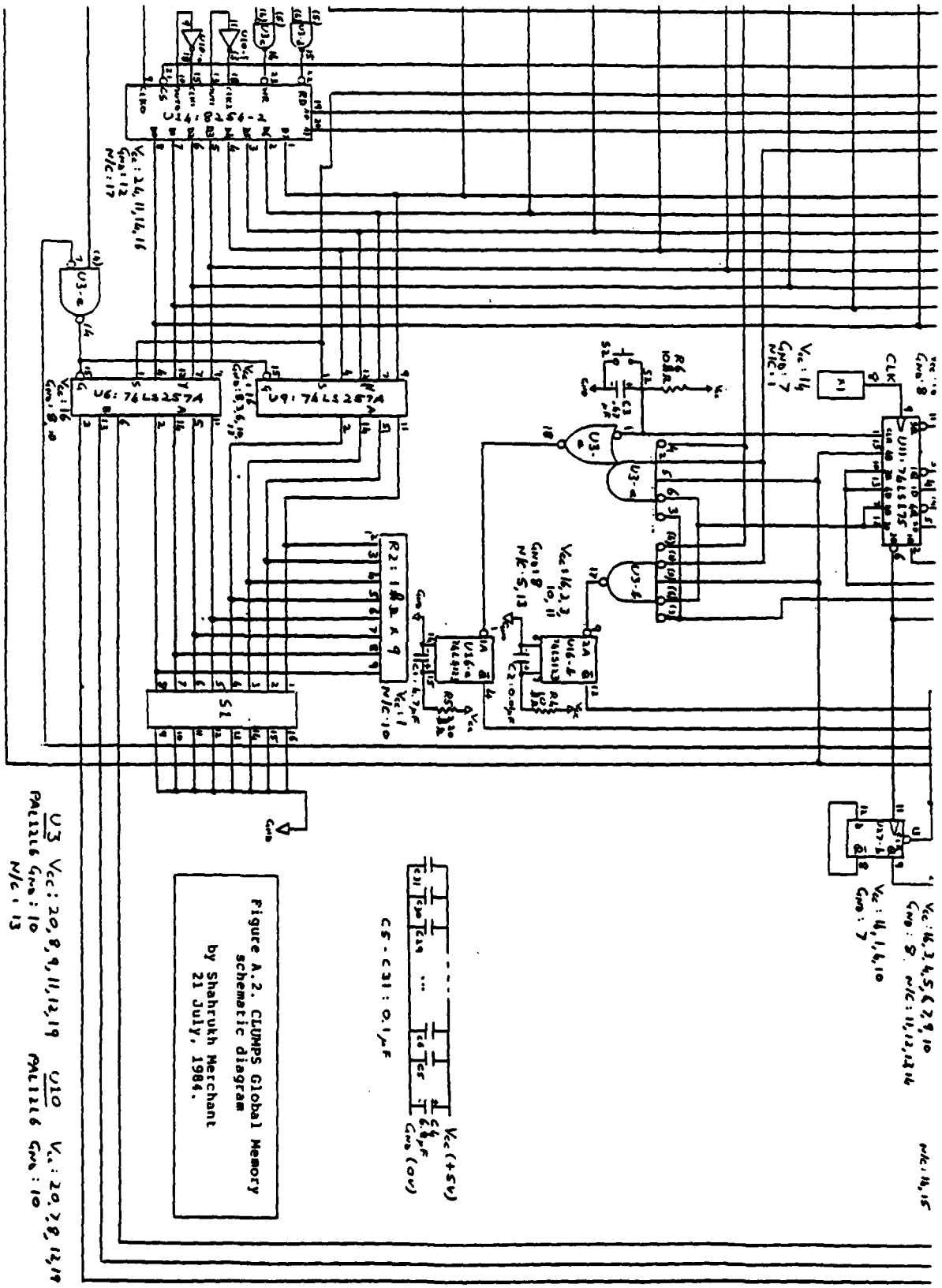


Figure A.2. CLUMPS Global Memory schematic diagram

by Shahrukh Merchant
21 July, 1984.

A.2.2. Parts list

Table A.3 shows a list of all parts used on the global memory board. The design of the four Programmable Array

Part #	Description		# of pins
U1	PAL14L4	Global decoder PAL	20
U2,5,8,13, 15,18,19,24	HM4864-2	Hitachi 64K x 1-bit dynamic RAMs	8x16
U3,10	PAL12L6	PALs for "random" logic	2x20
U4	74LS245	Octal bus transceiver	20
U6,7,9,12	74LS257A	Quad 2-to-1 multiplexers	4x16
U11	74LS175	Quad D flip-flops	16
U14	Intel 8254-2	Programmable Timer	24
U16	74LS123	Dual monostables	16
U17	74LS244	Octal buffers	20
U20	74LS374	Octal D flip-flops	20
U21	74LS30	8-input NAND	14
U22	74LS161A	4-bit binary counter	16
U23	74LS393	Dual 4-bit binary counters	14
U25	PAL16R8	Arbiter PAL	20
U26	74LS148	Octal priority encoder	16
U27	74LS74	Dual D flip-flops	14
X1	LOCO II	Motorola 19.6608 MHz oscillator	4 (14)
S1	8 SPST DIP Switches		16
S2	SPST normally open push-button		2
R1,2,7	9 1kΩ bussed resistors		3x10
R3	8 27Ω resistors		16
R4	10 kΩ, 1/4 W resistor		2
R5	220 kΩ, 1/4 W resistor		2
R6	10 kΩ, 1/4 W resistor		2
R8,9,12	27 Ω, 1/4 W resistors		3x2
R10	150 Ω, 1/2 W resistor		2
R11	1 kΩ, 1/4 W resistor		2
C1	4.7 μF tantalum capacitor		2
C2	0.01 μF capacitor		2
C3	0.47 μF capacitor		2
C4	6.8 μF tantalum capacitor		2
C5-31	0.1 μF bypass capacitors		27x2

Table A.3. CLUMPS Global memory board parts list

Logic devices (PALs) are presented in Sections A.4 and A.5.

A.3. Global memory board operation

A.3.1. Overview of circuit operation

U11, a 74LS175 consisting of four D-type flip-flops configured as a 4-bit twisted-ring counter, is the basic timing element and generates the various timing waveforms required. It is driven by X1, a 19.6608 MHz crystal oscillator. It has eight valid states, labelled *a* through *h*, which are shown in Table A.4, together with four of the available outputs. The complements of these outputs are also available, but are not shown in the table. A memory cycle starts at the beginning of state *a* and ends at the end of state *h*. The basic memory cycle time is, therefore,

$$\frac{8}{19.6608\ \text{MHz}} = 406.90\ \text{ns} \approx 400\ \text{ns}.$$

States:		*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*
OUTPUTS	1Q	0	1	1	1	1	0	0	0
	2Q	0	0	1	1	1	1	0	0
	3Q	0	0	0	1	1	1	1	0
	4Q	0	0	0	0	1	1	1	1

Table A.4. Global memory waveform generation

U20 latches the requests at the end of state *f* (about 100 ns before the start of the cycle). At the start of state *a*, U25, the arbiter, removes its previous acknowledgement and drives a new acknowledgement line low if

one or more request was pending.

Any time that there is no request for a particular cycle, that cycle is used for memory refresh. NAND gate U21 effectively ORs together all the requests to determine whether any request is pending. Flip-flop U27-a keeps track of whether the current cycle is a refresh cycle or a "normal" cycle (i.e., not a refresh cycle).

Flip-flop U27-b and 4-bit counter U22 together form a 5-bit counter which counts the number of cycles since the last refresh. They force a refresh every 32 cycles if none of the previous 32 cycles were idle. This ensures that a refresh is done at least every 32x407 μs, which means that a total refresh of all 128 rows of the dynamic memory is done no less frequently than every 128x32x407 μs = 1.67 ms, well within the 2.0 ms specified for the Hitachi HM4864-2 memories [HIT80]. 8-bit counter U23 (74LS393) is the refresh address counter which increments at the end of every refresh cycle. It places its count on the memory address bus via buffer U17 (74LS244), instead of the regular address inputs, at the appropriate time in the refresh cycle.

Note that the clock to the arbiter is disabled by gate U10-c during refresh cycles, and that the outputs of the arbiter are enabled only during "normal" cycles when no request are pending (they are pulled up to +5V and so are high otherwise).

The memories are all 64K x 1-bit dynamic RAMs.

Multiplexers U7 and U12 (74LS257A) place the high- and low-order bytes of the address (row and column address, respectively) on the memory address bus at the appropriate points in the cycle. 27Ω resistors R3, R8, R9 and R12 prevent excessive ringing on the capacitive memory inputs. U4 is a bidirectional data buffer. U1 is a global-memory address decoder implemented on a PAL (see Section A.4.1).

U14 is an Intel 8254-2 programmable counter [INT82] cascaded into a single 48-bit counter which increments every memory cycle (406.9 $n\delta$). It can be programmed by any branch processor and can be used as a real-time clock. It is not presently configured to automatically generate interrupts. Monostable multivibrators U16-a and U16-b generate RESETOUT and NMIOUT pulses, respectively, when the appropriate addresses are written. Resistor and capacitor values are chosen to yield a RESETOUT pulse width of about 500 $m\delta$ and an NMIOUT pulse width of about 50$\mu\delta$.

A.3.2. Timing diagrams

Timing diagrams for the global memory board are shown in Figure A.3 below, and include dynamic memory signals, 8254 counter timing signals, request and acknowledgement generation, refresh cycles, etc. The following signals are shown in Figure A.3, and are identified in the schematic diagram, Figure A.2.

CLK: Local clock, 19.6608 MHz

$\overline{\text{RAS}}$: Row address strobe for memory (local)

$\overline{CAS}$: Column address strobe for memory (local)

$\overline{ACK}_i$: Acknowledge to ith branch (global output)

$\overline{REQ}_i$: Request from ith branch (global input)

A_{0-15}: Address bus (global inputs)

D_{0-7}: Data bus (global bidirectional)

R/$\overline{W}$: Read/write signal (global input)

R/$\overline{C}$: Row/column memory control signal (local)

MEMADDR: High or low byte of address applied to dynamic RAMs (local)

N/$\overline{R}$: Indicates whether current cycle is normal or refresh

REFREQ: Indicates request for forced refresh cycle (local)

REQ: High if any request line is active (local)

ARBCLK: Clock to arbiter (U25) (local)

$\overline{ARBCE}$: Arbiter (U25) chip enable signal (local)

$\overline{WR}$: 8254 write-enable input (local)

$\overline{RD}$: 8254 read-enable input (local)

The designation "local" refers to a signal on the global memory board which does not appear on the bus; "global" means that it is a global bus signal.

A.4. PAL implementation of logic

For flexibility and to conserve board space, the global memory decoder, all "random" logic and the arbiter were programmed onto Programmable Array Logic (PAL) devices [NAT82]. The design of the arbiter is presented in Section A.5; the decoder and random logic are shown below.

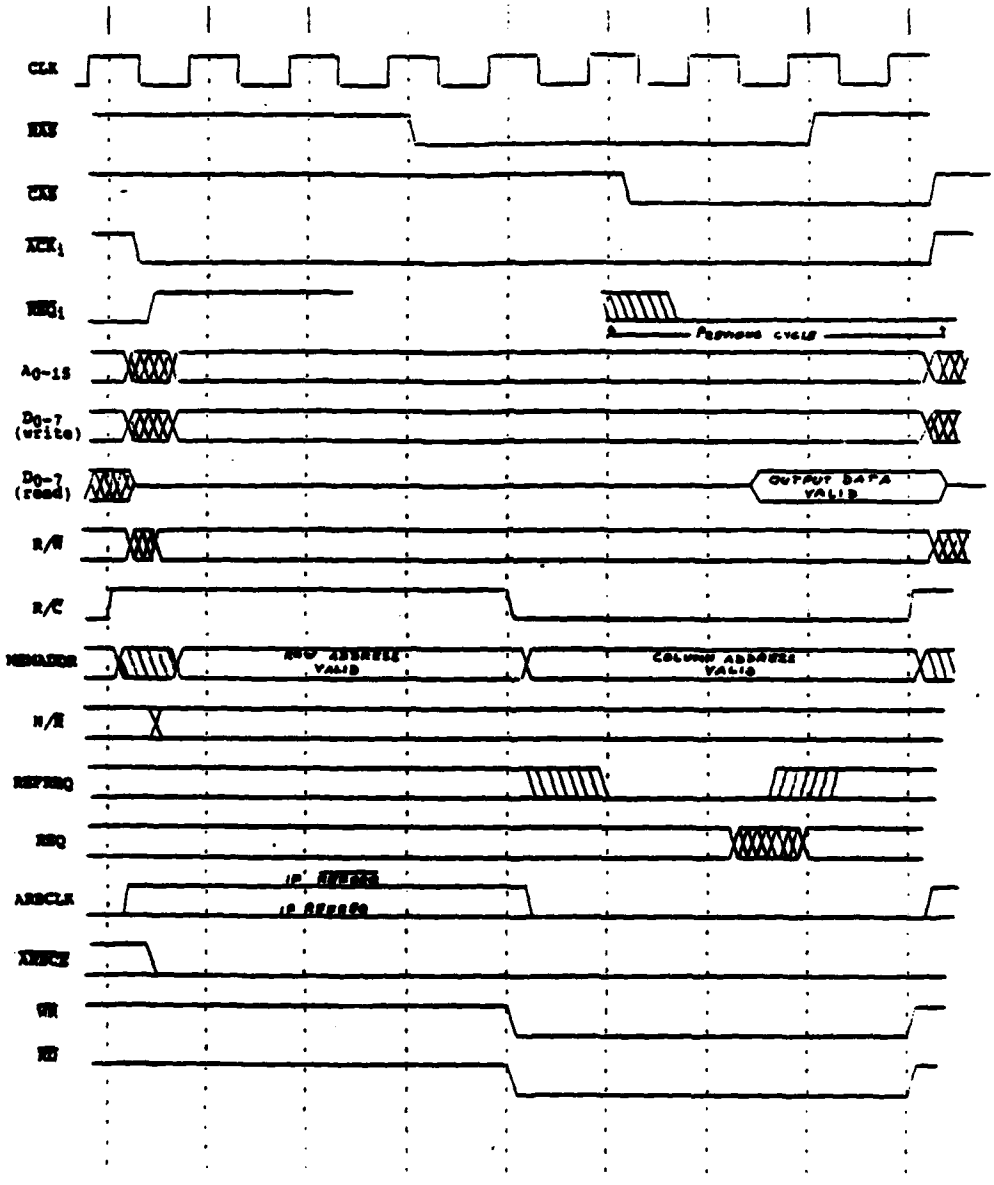


Figure A.3. Timing waveform for global memory board

A.4.1. Bus address decoder

A PAL14L4 device is used to implement the decoder (U1). The decoder outputs are ID (node or branch ID), CNTR (8254 counter), RESET (reset or NMI) and RAM (global RAM). From the global memory map (Table A.2) we form the following signals by decoding the upper 13 bits of the address bus. The relevant equations are:

$$ID = (N/\overline{R})\ [\bigwedge_{i=3}^{6}\overline{A}_i]\ A_7\ [\bigwedge_{i=8}^{15}\overline{A}_i]$$

$$CNTR = (N/\overline{R})\ A_3\ [\bigwedge_{i=4}^{6}\overline{A}_i]\ A_7\ [\bigwedge_{i=8}^{15}\overline{A}_i]$$

$$RESET = (N/\overline{R})\ A_4\overline{A}_5\overline{A}_6A_7\ [\bigwedge_{i=8}^{15}\overline{A}_i]$$

$$\overline{RAM} = (\overline{N/\overline{R}}) + [\bigwedge_{i=4}^{6}\overline{A}_i]\ A_7\ [\bigwedge_{i=8}^{15}\overline{A}_i]$$

The schematic diagram is shown in Figure A.4.

A.4.2. Miscellaneous logic

The following other signals are implemented on two PAL12L6 devices, U3 and U10. The signal names used are usually identified on the schematic diagram (Figure A.2). The signals 1Q, 2Q, 3Q and 4Q refer to the outputs of U11. All other signals are identified by the device number and pin function of a component to which it is connected and which indicates its primary function (e.g., U14-RD).

The following equations are implemented on U3:

a) $U16\text{-}1A = \overline{S2} + A_3 \cdot RESET \cdot \overline{R/\overline{W}} \cdot 4Q \cdot \overline{2Q}$

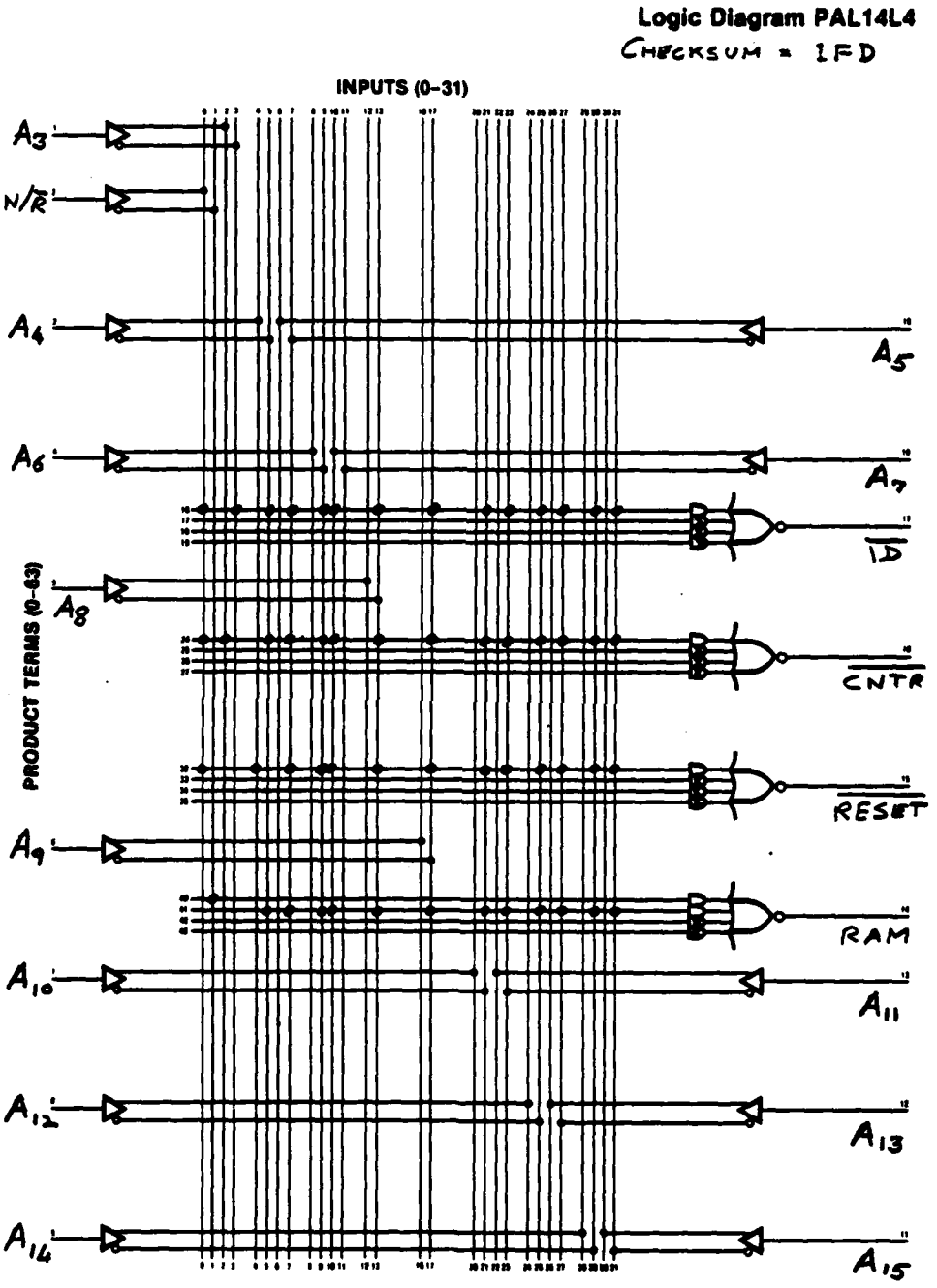


Figure A.4. Global memory decoder PAL

b) U16-2A = $\overline{A_3}$ · RESET · $\overline{R/\overline{W}}$ · 4Q · $\overline{2Q}$

c) U14-WR = 4Q · $\overline{R/\overline{W}}$

d) U14-RD = 4Q · $R/\overline{W}$

e) U6/9-G = $R/\overline{W}$ · ID

The following equations are implemented on U10:

a) U14-CLK1 = $\overline{\text{U14-OUT0}}$

b) CAS = RAM · $N/\overline{R}$ · 4Q · $\overline{IQ}$

c) ARBCLK = $\overline{4Q + REFREQ}$

d) $N/\overline{R}$NEXT = REQ · $\overline{REFREQ}$

e) U23-CLK (= U22-CLR = U27b-CLR) = $\overline{N/\overline{R}}$ · 4Q

f) U14-CLK2 = $\overline{\text{U14-OUT1}}$

The schematic diagrams for U3 and U10 are shown in Figures A.5 and A.6 respectively.

A.5. PAL implementation of arbiter

Section 2.3 describes the operation of the arbiter. Equation 2.1 from that section is re-written below:

$$Ack_i' = Req_i \bigvee_{j=1}^{8} [Ack_{i-j} (\bigwedge_{k=1}^{j-1} \overline{Req_{i-k}})] \qquad (2.1)$$

As pointed out in that section, there are two problems with this equation as it stands, even though it is correct in principle. These are outlined below, and a modified equation is obtained, which can be implemented on a single PAL16R8 Programmable Array Logic device.

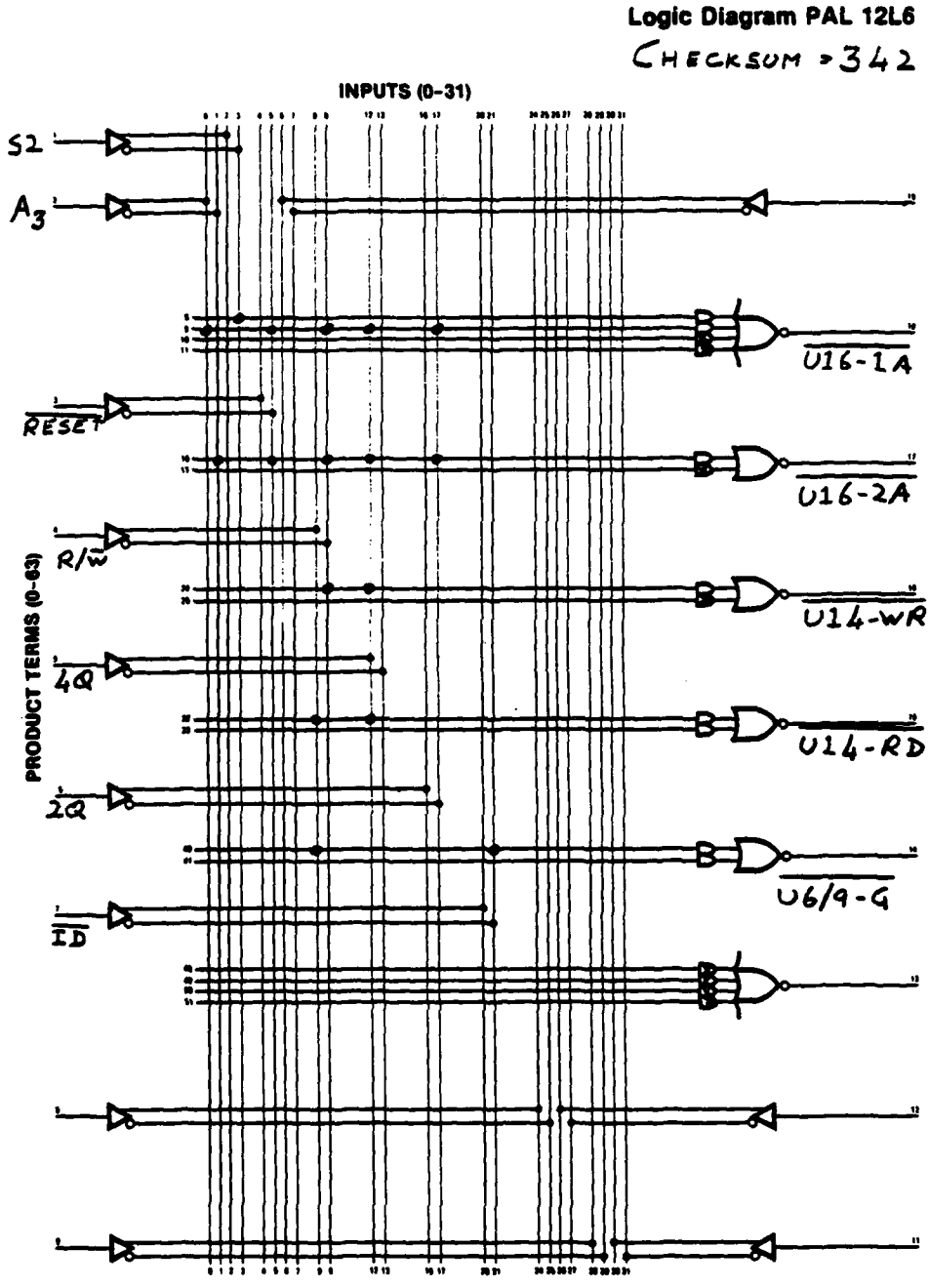


Figure A.5. Schematic diagram for "random" logic implemented on U3

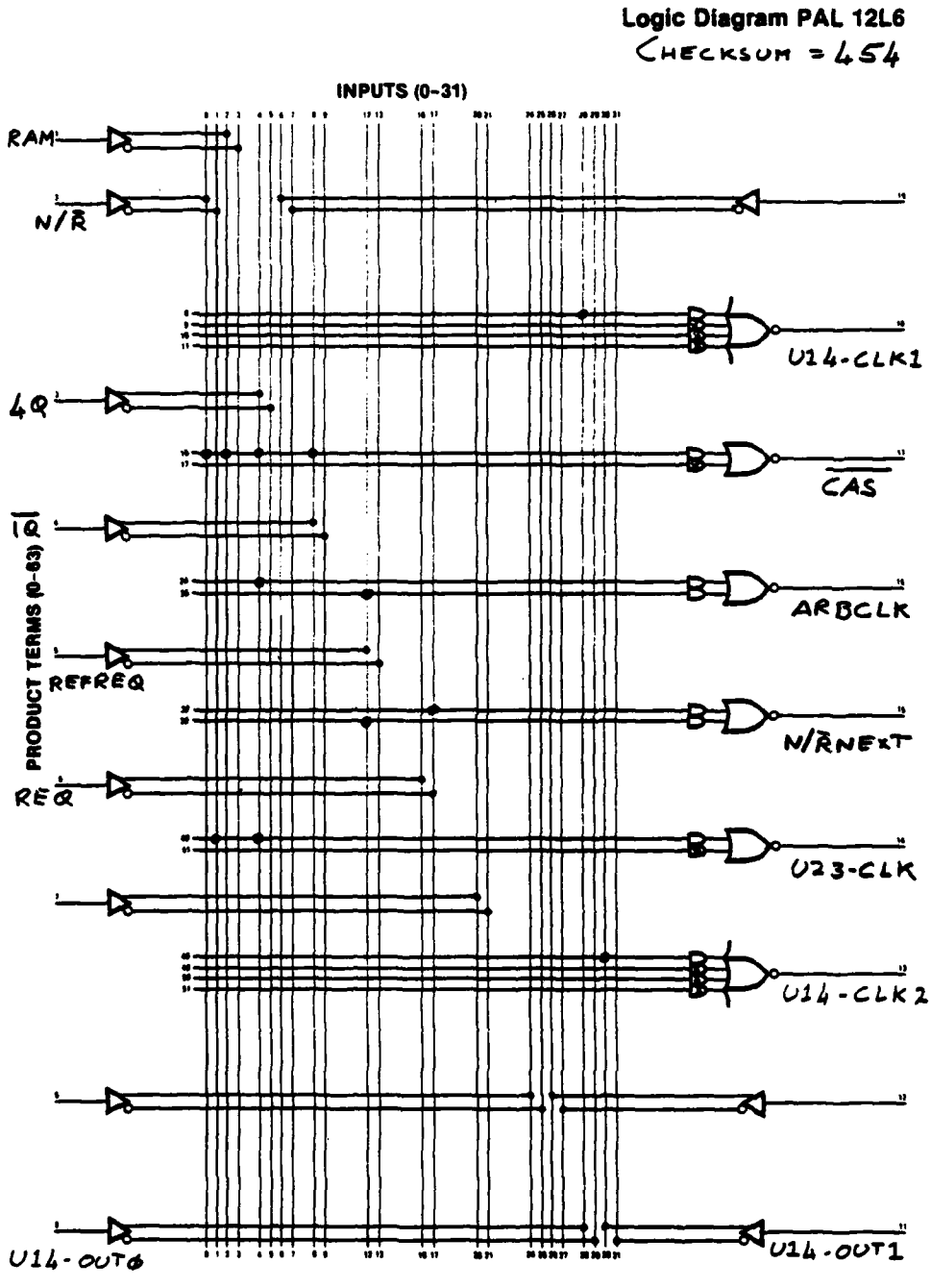


Figure A.6. Schematic diagram for "random" logic implemented on U10

A.5.1. Fair arbiter

The first problem is to have the flip-flops associated with the Ack_is in the arbiter retain their states even when no requests are pending.(1) If this were not done, all the Ack_is would be low and would always stay that way if Equation 2.1 were used directly. One possibility is to disable the arbiter clock during idle cycles, but a simpler solution is to OR in one extra term to Equation 2.1 for Ack_i', namely:

$$Ack_i\overline{Req}_{i-7}\overline{Req}_{i-6} \ldots \overline{Req}_{i-1}\overline{Req}_i$$

which, in effect, says: "Retain present state if no requests are active."

Observe, now, that the last term in Equation 2.1 is

$$Ack_i\overline{Req}_{i-7}\overline{Req}_{i-6} \ldots \overline{Req}_{i-1}Req_i$$

which, when ORed with the extra term above simplifies to

$$Ack_i\overline{Req}_{i-7}\overline{Req}_{i-6} \ldots \overline{Req}_{i-1} .$$

Thus, Equation 2.1 is modified to:

$$Ack_i' = Req_i \bigvee_{j=1}^{7} [Ack_{i-j} (\bigwedge_{k=1}^{j-1} \overline{Req}_{i-k})] + Ack_i (\bigwedge_{k=1}^{7} \overline{Req}_{i-k}) \quad (A.1)$$

This solves the above-mentioned problem. The other problem occurs during power-up; we have to ensure that the

---

(1) The output control on the arbiter ensures that when no requests are pending, all Ack_is are inactive externally even though one of the flip-flops internal to the arbiter is set.

arbiter initializes in a valid state, i.e., exactly one of the flip-flops representing the Ack_is should be set. This problem is solved by replacing Ack_i in Equation A.1 by A_i, where

$$A_i = \begin{cases} \overline{Ack}_0\overline{Ack}_1 \ldots \overline{Ack}_{i-1}Ack_i & (i=0,1,\ldots,6) \\ \overline{Ack}_0\overline{Ack}_1 \ldots \overline{Ack}_6 & (i=7) \end{cases} \quad (A.2)$$

It is easily seen that in a valid state (exactly one Ack_i set), A_i and Ack_i are always equal. In an invalid state, it behaves as follows: If all Ack_is are logic 0, A_7 is high and all the other A_is are low. If more than one Ack_i is logic 1, only the A_i corresponding to the lowest numbered one will be high. Note that after the very first clock cycle, the arbiter will immediately enter a valid state and will remain in valid states thereafter.

Thus, the final equation for the PAL is:

$$Ack_i' = Req_i \bigvee_{j=1}^{7} [A_{i-j} \left(\bigwedge_{k=1}^{j-1} \overline{Req}_{i-k}\right)] + A_i \left(\bigwedge_{k=1}^{7} \overline{Req}_{i-k}\right) \quad (A.3)$$

where A_i is as given by Equation A.2. Figure A.7 shows the schematic diagram for the arbiter implemented on a PAL16R8.

A.5.2. Original arbiter design

Sections 2.3.1 and 3.6 refer to the "biased arbiter" which was discarded in favour of the design in Section A.5.1 above. For reference, we give the equations corresponding to the biased arbiter:

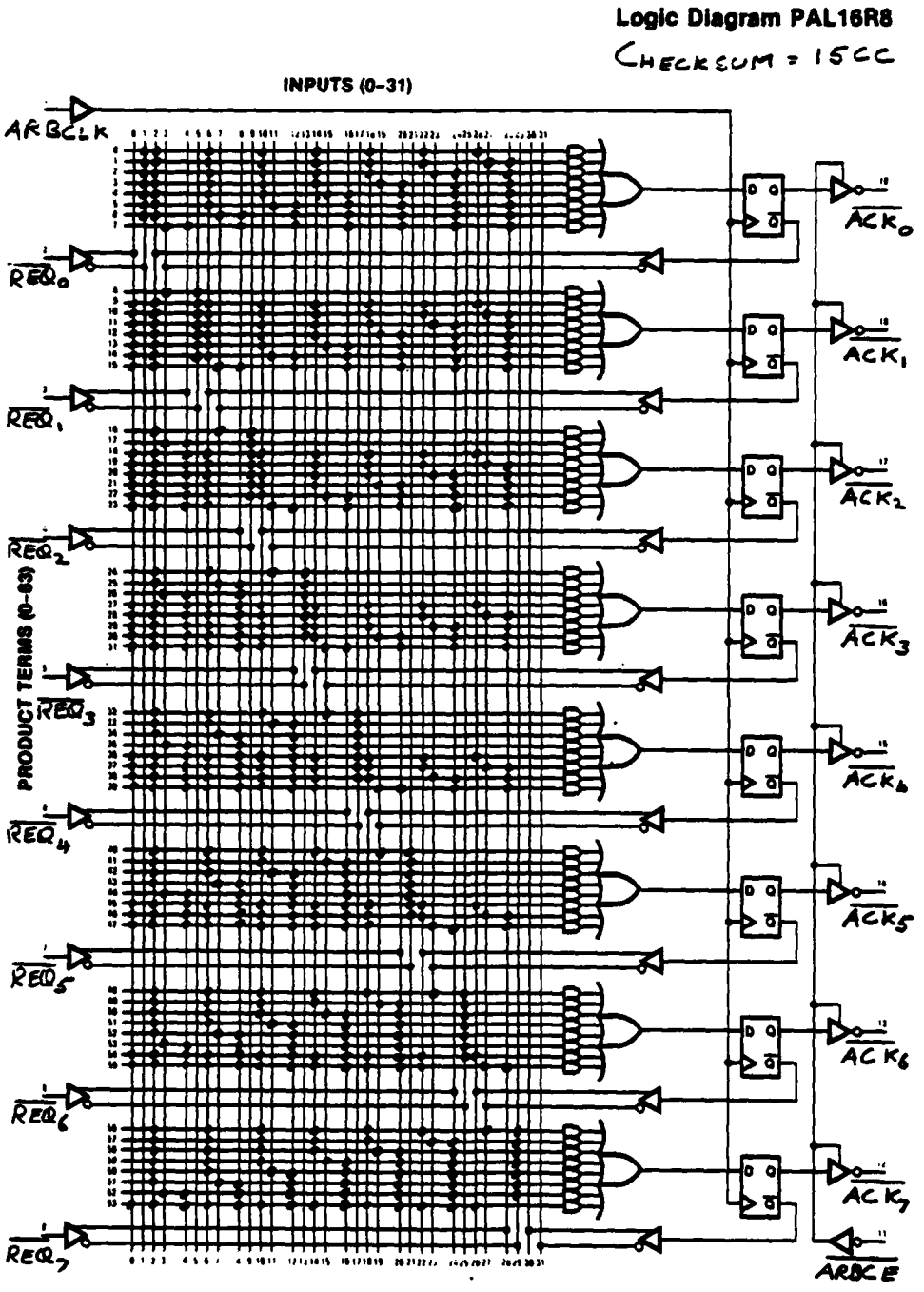


Figure A.7. Schematic diagram for arbiter PAL

$$Ack_i' = Req_i \bigvee_{j=1}^{8} [A_{i-j} (\bigwedge_{k=1}^{j-1} \overline{Req}_{i-k})]$$

with

$$A_i = \begin{cases} Ack_i & (i=0,1,\ldots,6) \\ \overline{Ack}_0\overline{Ack}_1 \ldots \overline{Ack}_6 & (i=7) \end{cases}$$

When no devices were making requests, all flip-flops would be reset and all A_is would be logic 0, except A_7, which would be logic 1. Thus, because of the round-robin system, lower numbered devices would have higher priority in the event that two devices subsequently generated requests simultaneously.

B. ANALYSIS AND SIMULATION SOFTWARE

B.1. Listing of program to calculate parameters for FCFS model

```
{$R+}
program Fifo_arbiter;
{
   Author: Shahrukh Merchant
   Date: 14 May, 1984
   Last revised: 21 May, 1984

   Language: "Turbo" Pascal Version 1.00 on the IBM Personal Computer

   This program models the arbiter as a FIFO dicrete-time queue, and computes
   the steady-state probability mass functions for (a) the number of customers
   (devices that have requested service) in the system and the waiting time
   (including service) of a customer.

   Important variables used are:

   N (constant) = Total number of devices in the system being simulated.
   NP1 (constant) = N + 1.
   EPSILON (constant) = Magnitude of largest number that will be treated as
                        zero (due to round-off, etc.).
   p = probability of any inactive device making a request in a time slot
       (0<p<1).
   pi[i] = steady-state probability that i devices are active (i customers are
           in the system.
   tableau = transition matrix for system, augmented with an extra row and
             column, and adjusted so as to facilitate solving the matrix
             equation   pi = pi * P.  If i and j are the initial and final
             states respectively, then let P[i,j] = Probability of going from
             i to j in 1 step.
             The system of equations we need to solve is:
                    pi * [P - I]   = 0    or   [P - I](t) * pi = 0
                pi * [1 ... 1](t) = 1.        [1 ... 1] * pi  = 1

                  (The (t) symbol denotes the matrix transpose.)

             So "tableau" has the form:

                                          |    (t)      | 0 |
                                          |   P - I     | : |
                                          |             | 0 |
                                          |-------------|---|
                                          | 1 1 1 . . . 1 | 1 |

             and one of the equations is redundant.

   choose[i,j] = binomial coefficient (i,j) for 0 <= i,j <= N  (it is zero
                 for undefined combinations, e.g., (3,4)).
   p_exp[i] = i'th power of p, for 0 <= i <= N.
   cp_exp[i] = i'th power of (1-p), for 0<= i <= N.
```

```
   wait_pdf[i] = Probability that an arriving customer has to wait for time i
                 (including service time).
}

{
 Begin main program declarations
}

const N = 8;
      NP1 = 9;
      EPSILON = 1.0e-8;

var   p, average, mean_square :real;
      pi                      :array[0..N] of real;
      tableau                 :array[0..NP1,0..NP1] of real;
      choose                  :array[0..N,0..N] of real;
      p_exp, cp_exp           :array[0..N] of real;
      i                       :integer;
      wait_pdf                :array[1..N] of real;
      answer                  :char;
{
 Begin subprogram declarations
}
   procedure print_tableau;
   {
      This is a DEBUGging procedure and simply prints out a dump of the
      array tableau, row by row.
   }
   var i,j   :integer;

   begin
   for i:= 0 to N+1 do
      begin
      for j:= 0 to N+1 do write (tableau[i,j]);
      writeln
   end
   end;  {procedure print_tableau}


   procedure initialize;
   {
      This procedure initializes the following output variables/arrays
      (declared globally) to their required values.

      Outputs: choose    p_exp    cp_exp    tableau

      Inputs: N, EPSILON, p
   }
   var i,j     :integer;
       total   :real;    {for DEBUGging only}

   begin
   {
```

```
 Initialize "choose"
}
for i:= 0 to N do
                for j:= 0 to N do choose[i,j]:= 0;
choose[0,0]:= 1;
for i:= 1 to N do
   begin
   choose[i,0]:= 1;
   for j:= 1 to N do choose[i,j]:= choose[i-1,j-1] + choose[i-1,j]
end;
{
 Initialize p_exp and cp_exp
}
p_exp[0]:= 1; cp_exp[0]:= 1;
for i:= 1 to N do
   begin
   p_exp[i]:= p_exp[i-1] * p;
   cp_exp[i]:= cp_exp[i-1] * (1-p)
end;
{
 Initialize tableau
}
for i:= 0 to N do
                for j:= 0 to N+1 do tableau[i,j]:= 0;
for j:= 0 to N+1 do tableau[N+1,j]:= 1;
{Fill in non-zero transition probabilities}
for j:= 0 to N do tableau[j,0]:= choose[N,j] * p_exp[j] * cp_exp[N-j];
                                                        {row 0}
for i:= 1 to N do
          for j:= i-1 to N-1 do tableau[j,i]:= choose[N-i,j-i+1] *
                                   p_exp[j-i+1] * cp_exp[N-j-1]; {other rows}

{DEBUG CODE BEGIN} {Check for stochastic matrix}
for i:= 0 to N do
   begin
   total:= 0;
   for j:= 0 to N do total:= total + tableau[j,i];
   if abs(total-1.0) >= EPSILON then
      begin
      writeln ('*** ERROR: Column', i:4, ' is not stochastic.',
               ' Tableau reads:');
      print_tableau; halt
   end
end;
writeln ('* Stochastic test successful *');
{DEBUG CODE END}

{Subtract out identity matrix}
for i:= 0 to N do
   begin
   tableau[i,i]:= tableau[i,i] - 1
end
end; {procedure initialize}
```

```
procedure solve;
{
   This procedure carries out Gaussian elimination with back substitution
   on "tableau" to obtain the pi vector.  Scaled partial pivoting is used
   to reduce round-off error.  See S.D. Conte & Carl de Boor, "Elementary
   Numerical Analysis: An Algorithmic Approach," McGraw-Hill, 1972.

   Inputs: N, NP1, EPSILON, tableau

   Outputs: pi, tableau (destroyed)
}
var size                  :array[0..NP1] of real; {used in pivoting strategy}
    index, i, j, k        :integer;
    factor, max, temp     :real;

begin
for k:= 0 to N do   {for each variable}
   begin
   for i:= k to N+1 do
      begin
      temp:= 0;
      for j:= k to N do temp:= temp + abs(tableau[i,j]);
      size[i]:= temp
   end;
   max:= 0; index:= -1;
   for i:= k to N+1 do
      begin
      temp:= abs(tableau[i,k]);
      if size[i] >= EPSILON then
                  if temp/size[i] >= max then
                     begin
                     max:= temp/size[i];
                     index:= i
                  end
   end;

   {DEBUG CODE BEGIN}
   if index = -1 then
      begin
      writeln ('*** ERROR: Zero pivot element at step', k:4, 'Tableau is:');
      print_tableau; halt
   end;
   {DEBUG CODE END}

   {Exchange rows}
   if k <> index then
           for j:= k to N+1 do
              begin
              temp:= tableau[k,j];
              tableau[k,j]:= tableau[index,j];
              tableau[index,j]:= temp
           end;
   for i:= k+1 to N+1 do
```

```
      begin
      factor:= tableau[i,k]/tableau[k,k];
      tableau[i,k]:= 0;
      for j:= k+1 to N+1 do
                  tableau[i,j]:= tableau[i,j] - factor * tableau[k,j]
   end
end; {k loop}

{DEBUG CODE BEGIN}
writeln ('* Pivoting complete *');
{Verify that one equation was redundant (last row 0)}
if abs(tableau[N+1,N+1]) >= EPSILON then
   begin
   writeln ('*** ERROR: Redundant equation inconsistent. Tableau reads:');
   print_tableau; halt
   end
else writeln ('* One redundant equation. O.K. *');
{DEBUG CODE END}

{
 Back substitution to get pi vector
}
for i:= N downto 0 do
   begin
   temp:= 0;
   for j:= i+1 to N do temp:= temp + tableau[i,j] * pi[j];
   pi[i]:= (tableau[i,N+1] - temp)/tableau[i,i]
end;

{DEBUG CODE BEGIN}
{Recreate transition matrix (use tableau for storage) and ensure that
 pi=pi*P is satisfied and that the pi[i]'s sum to 1.}

{Check pi[i]'s sum to 1}
temp:= 0;
for i:= 0 to N do temp:= temp + pi[i];
if abs(temp-1.0) >= EPSILON then
   begin
   writeln ('*** ERROR: pi[i]''s do not sum to unity. pi vector is:');
   for i:= 0 to N do write (pi[i]);
   writeln; halt
   end
else writeln ('* pi[i]''s sum to unity. O.K. *');

{Recreate transition matrix}
for i:= 0 to N+1 do
                  for j:= 0 to N+1 do tableau[i,j]:= 0;
for j:= 0 to N do tableau[0,j]:= choose[N,j] * p_exp[j] * cp_exp[N-j];
for i:= 1 to N do
         for j:= i-1 to N-1 do tableau[i,j]:= choose[N-i,j-i+1]
                                * p_exp[j-i+1] * cp_exp[N-j-1];
{Check for pi=pi*P}
for j:= 0 to N do
```

```
   begin
   temp:= 0;
   for i:= 0 to N do temp:= temp + pi[i] * tableau[i,j];
   if abs(pi[j]-temp) >= EPSILON then
      begin
      writeln ('*** ERROR: pi=pi*P not satisfied. pi vector is:');
      for i:= 0 to N do write (pi[i]);
      writeln; writeln ('Transition matrix is:');
      print_tableau; halt
   end
end;
writeln ('* pi=pi*P is O.K. *')
{DEBUG CODE ENDS}

end; {procedure solve}


procedure waiting_time;
{
   This procedure computes the waiting time probability mass function.
   The waiting time will always be in the range 1 to N inclusive.

   Inputs: pi, N, choose, p_exp, cp_exp, p, EPSILON (DEBUG only)

   Outputs: wait_pdf
}
var i,s,g,j   :integer;
    sum       :real;

begin
for i:= 1 to N do
   begin
   wait_pdf[i]:= pi[0] * choose[N,i] * p_exp[i] * cp_exp[N-i];
   for s:= 0 to i do
      begin
      sum:= 0;
      for g:= i-s+1 to N-s do sum:= sum + choose[N-s,g] * p_exp[g]
                                        * cp_exp[N-s-g];
      wait_pdf[i]:= wait_pdf[i] + pi[s] * sum
   end;

   sum:= 0;
   for j:= 0 to N-1 do sum:= sum + pi[j] * (N-j);
   wait_pdf[i]:= wait_pdf[i]/(p*sum)
end;

{DEBUG CODE BEGIN}
{Check that probabilities sum to 1}
sum:= 0;
for i:= 1 to N do sum:= sum + wait_pdf[i];
if abs(sum-1.0) >= EPSILON then
   writeln ('*** ERROR: Waiting-time probabilities do not sum to 1.')
```

```
   else writeln ('* Waiting time probabilities sum to 1. O.K. *')
   {DEBUG CODE ENDS}

   end; {procedure waiting_time}

{
 BEGIN MAIN PROGRAM
}
begin
repeat
   write ('Enter request probability: '); readln (p);
   initialize;

   solve;  {solve tableau for steady-state probability vector pi}
   writeln; writeln ('Steady-state probability vector is:'); writeln;
   average:= 0;
   for i:= 0 to N do
      begin
      writeln ('pi[', i:2, '] = ', pi[i]);
      average:= average + i * pi[i]
   end;
   writeln; writeln ('Average number in system =', average);

   waiting_time; {compute waiting-time probability mass function}
   writeln; writeln ('Waiting time probability vector is:'); writeln;
   average:= 0; mean_square:= 0;
   for i:= 1 to N do
      begin
      writeln ('w[', i:2, '] = ', wait_pdf[i]);
      average:= average + i * wait_pdf[i];
      mean_square:= mean_square + sqr(i) * wait_pdf[i]
   end;
   writeln; writeln ('Average waiting time =', average, ',   Variance =',
                     mean_square - sqr(average));

   write ('Restart (R) or Quit (Q) ? ');
   readln (answer); answer:= upcase(answer)
until answer <> 'R'

end. {program Fifo_arbiter}
```

B.2. Simulator listing

```
{$R+}
program Arbiter_simulate;
{
   Author: Shahrukh Merchant
   Date: 21 May, 1984
   Last revised: 21 May, 1984
   Also revised: 21 June, 1984 to include "Fair" arbiter
   Revised 24 June, 1984 to include unequal request probability for device 1.
   Revised 25 June, 1984 to allow general file I/O.
   Revised 2 July, 1984 to allow variable (integer) idle_len (formerly
      always 1).
   Revised 6 July, 1984 to include First-come, First-served (FCFS) arbiter.

   Language: "Turbo" Pascal Version 1.00 on the IBM Personal Computer

   This program simulates the arbiter as a discrete time round-robin
   queue, to determine the probability mass functions of:
     1) the number of customers in the system
     2) the waiting time of a customer
     3) the length of a busy period and
     4) the length of an idle period.

   Three types of arbiters are implemented:
     1) A "Fair" round-robin arbiter
     2) A "Biased" round-robin arbiter and
     3) A First-come, First-served arbiter.
   See the discussion in the text of procedure next_device for a description
   of the first two.  The third is a simple FCFS queue, with random selection
   in the case of simultaneous arrivals.

   The waiting time distribution is determined separately for each customer
   as well as for the whole system.

        Request generation can be a function of a) the device number and
   b) the number of cycles since the device was last served (including 0).
   Therefore, it does not have to be the same for all devices, nor does it
   have to have the same constant probability per cycle for every inactive
   device.  However, a device that has a request pending cannot generate
   another request.  The request generation is governed by the function
   "pr_req" and the procedure "init_pr_req."

   Important variables used are:
   infile, outfile = Input & output file variables, respectively.
   N (constant) = Total number of devices in the system being simulated.
   MAX_BUSY, MAX_IDLE (constants) = Maximum busy and idle period lengths,
        respectively, that we want to keep track of in their respective
        distributions.
   clock = count of number of cycles of simulation (a real number since
        integers only go up to 32767.
   dev_idle[i] = length of current idle period for device i (set to zero
        if not idle).
```

```
n_cust = Number of customers currently in system (making requests or
     being served).
request[i] = "true" if device i is currently making a request, "false"
     otherwise.
current_dev = device currently being served (0 if no device is being
     served).
last_dev = device currently being served, or last device served if no
     device currently being served.  Used only for "Fair" arbiter.
dev_wait[i] = current waiting time for device i (0 if device i is not
     currently requesting).
wait_pdf[i,j] = number of times during simulation that device i waited
     for time j (1 <= i,j <= N).
n_cust_pdf[i] = number of simulation cycles that the number of customers
     in the system was i.
busy_time = number of cycles that system has been continuously busy (0 if
     system idle).
idle_time = number of cycles that system has been continuously idle (0 if
     system busy).
idle_len = number of cycles after a request for a particular device has
     been served that that device is guaranteed to be idle (i.e., will
     not generate a request).
busy_pdf[i] = number of busy periods of length i (1 <= i <= MAX_BUSY).
     busy_pdf[0] is used to store the number of busy periods of length
     > MAX_BUSY.
idle_pdf[i] = number of idle periods of length i (1 <= i <= MAX_IDLE).
     idle_pdf[0] is used to store the number of idle periods of length
     > MAX_IDLE.
longest_busy, longest_idle = longest busy and idle periods, respectively,
     encountered during simulation.
n_busy, n_idle = number of busy and idle periods, respectively,
     encountered during simulation.
dev_req[i] = number of requests serviced for device i during simulation.
iter_limit = maximum number of iterations for which simulation is to run.
busy_limit = maximum number of busy periods for which simulation is to run.
arbiter_type = 'F' if "Fair" arbiter is being used, 'B' if "Biased"
     arbiter is being used, 'I' if FCFS arbiter is being used.
queue[0..N] = array containing queue of request arrivals.  It can contain
     at most N elements (i.e., if all devices are requesting).  Used only
     for "FCFS" arbiter.
queue_head = points to top-most element of "queue."  Used for "FCFS"
     arbiter only.
queue_tail = points to element following bottom-most element of "queue."
     If queue is empty, queue_head=queue_tail.  Used for "FCFS" arbiter
     only.

sum_busy, sum_idle = sum ofalengths of all busy and idle periods,
     respectively, during simulation.
sum_busy_sq, sum_idle_sq = sum of square of lengths of all busy and idle
     periods, respectively, during simulation.
(These last 4 variables are used to determine the mean and variance of
 busy and idle period lengths.  This cannot be determined exactly from
 their probability mass functions busy_pdf and idle_pdf because the
```

```
    latter are truncated beyond MAX_BUSY and MAX_IDLE respectively.)
}

{
 Begin main program declarations
}

const N = 8;
      MAX_BUSY = 500;
      MAX_IDLE = 500;

type range = 0..N;
     small_range = 1..N;

var infile, outfile                                          :text;
    n_cust, current_dev, last_dev, queue_head, queue_tail    :range;
    idle_len                                                 :integer;

    p1, p, clock, busy_time, idle_time, n_busy, n_idle,
    sum_busy, sum_idle, sum_busy_sq, sum_idle_sq,
    iter_limit, busy_limit, longest_busy,
    longest_idle                                 :real;

    dev_idle, dev_req                            :array[1..N] of real;
    request                                      :array[1..N] of boolean;
    dev_wait                                     :array[1..N] of range;
    n_cust_pdf                                   :array[0..N] of real;
    busy_pdf                                     :array[0..MAX_BUSY] of real;
    idle_pdf                                     :array[0..MAX_IDLE] of real;
    wait_pdf                                     :array[1..N,1..N] of real;
    queue                                        :array[0..N] of small_range;

    answer, arbiter_type                         :char;

{
 Begin subprogram declarations
}
   procedure io;
   {
      Determine input and output files/devices
   }
   var filename      :string[14];

   begin
   write (con, 'Input file (use CON: for console input): ');
         {"con" is a pre-declared text-file variable for console I/O}
   readln (con, filename);
   assign (infile, filename);
   reset (infile);

   write (con, 'Output file (use CON: for console output): ');
   readln (con, filename);
   assign (outfile, filename);    rewrite (outfile)
```

```
end;   {procedure io}


procedure stop;
{
   Close files and halt when program terminates in error.
}
begin
close (infile);
close (outfile);
writeln (con, '*** Program terminated in error ***');
halt   {halt is a built-in function}
end;   {procedure stop}


procedure initialize;
{
   Initializes the following variables:

   clock, n_cust, current_dev, last_dev, busy_time, idle_time, n_busy,
   n_idle, longest_busy, longest_idle, sum_busy, sum_idle, sum_busy_sq,
   sum_idle_sq, dev_idle, request, dev_wait, wait_pdf, n_cust_pdf,
   dev_req, busy_pdf, idle_pdf, arbiter_type, queue, queue_head, queue_tail
}
var i,j        :integer;

begin
randomize;   {intrinsic function to initialize random number generator}

clock:= 0;
n_cust:= 0;
current_dev:= 0;
last_dev:= 8;
busy_time:= 0;
idle_time:= 0;
n_busy:= 0;
n_idle:= 0;
longest_busy:= 0;
longest_idle:= 0;
sum_busy:= 0;
sum_idle:= 0;
sum_busy_sq:= 0;
sum_idle_sq:= 0;
queue_head:= 0;
queue_tail:= 0;

n_cust_pdf[0]:= 0;
queue[0]:= 1;
for i:= 1 to N do
   begin
   dev_idle[i]:= 0;
   request[i]:= false;
   dev_wait[i]:= 0;
```

```
   n_cust_pdf[i]:= 0;
   queue[i]:= 1;
   dev_req[i]:= 0;
   for j:= 1 to N do wait_pdf[i,j]:= 0
end;

for i:= 0 to MAX_BUSY do busy_pdf[i]:= 0;
for i:= 0 to MAX_IDLE do idle_pdf[i]:= 0;

repeat
   write (con, 'Fair(F), Biased(B) or FCFS(1) arbiter ? ');
   readln (infile, arbiter_type); arbiter_type:= upcase (arbiter_type)
          {upcase is built-in function}
until arbiter_type in ['F','B','1']

end;   {procedure initialize}


procedure init_pr_req (var idle_len :integer; var p1, p :real);
{
   This procedure initializes all those parameters used by the function
   pr_req that are used to compute request probabilities.
   If these paramters are changed due to a change in the form of function
   pr_req, the following changes must be made to the rest of the program:
   1) Remove the declaration of all parameters that are no longer used
      from the declaration of variables in the main program, and from the
      comments describing the use of all variables.
   2) Add the declaration of all new parameters to the main program
      declarations, and to the comments describing the use of variables.
   3) Change the parameters of init_pr_req where it is invoked by the
      main program and elsewhere.
   4) Remove any other references to these parameters elsewhere in the
      program.

   Outputs: p (the probability that any inactive device (2-8) will make a
               request for any cycle other than the one immediately
               following a service completion of that device)
            p1 (same as above for device 1)
            idle_len (number of cycles after completion of a service for a
               device that that device is guaranteed to be idle).

}
begin
write (con, 'Enter number of guaranteed idle periods: ');
readln (infile, idle_len);
write (con, 'Enter device request probability (dev 1): ');
readln (infile, p1);
write (con, 'Enter device request probability (dev 2-8): ');
readln (infile, p)
end;   {procedure init_pr_req}
```

```
procedure input (var iter_limit, busy_limit :real);
{
   This procedure initializes those variables that control the running
   of the simulation.
}
begin
write (con, 'Maximum number of iterations ? ');
readln (infile, iter_limit);
write (con, 'Maximum number of busy periods ? ');
readln (infile, busy_limit)
end;   {procedure input}


function pr_req (dev_num :integer; dev_idle_time :real) :real;
{
   This function generates request probabilities greater than or equal
   to 0, and less than 1, for a device (number dev_num) to make a request
   for the next cycle, given that it has been idle for a certain number
   of cycles already (dev_idle_time).
        This function needs to be "customized" for different cases.
   Rewriting and recompiling can be avoided to a certain extent by using
   procedure init_pr_req to initialize parameters used by this function,
   if all one needs to do is to change parameters without changing the
   underlying structure.
        Note: While this procedure, and the simulation program in general,
   allows non-zero probabilities for a device for a slot (cycle)
   immediately following the one in which it got served, the actual
   hardware being simulated could never allow a request for a cycle to
   be generated while it is being serviced, so the same device could
   never be serviced for 2 consecutive cycles.

   Inputs: Device number, cycles since last request completed, p, N

}
begin

{DEBUG CODE BEGIN}
if (dev_idle_time <= -0.5) or not (dev_num in [1..N]) then
   begin
   writeln (outfile, '*** ERROR: invalid idle-time', dev_idle_time,
            ' for device', dev_num);
   stop
end;
{DEBUG CODE END}

if dev_idle_time <= idle_len - 0.5 then
   pr_req:= 0
else
   if dev_num=1 then
      pr_req:= p1
   else
      pr_req:= p
```

```
end;   {function pr_req}


procedure compute_requests;
{
   This procedure computes requests for every device, based on how
   long that device has been idle.
   For the FCFS arbiter, if simultaneous requests occur, they are
   randomly shuffled.

   Inputs: request, N, arbiter_type, queue, queue_head, queue_tail
   Outputs: request, dev_idle, queue, queue_tail
   Calls: pr_req
}
var i            :small_range;
    temp, count  :range;
    shuffle      :array[1..N] of small_range;

begin
count:= 0;
for i:= 1 to N do
      if not request[i] then
         begin
         request[i]:= random <= pr_req(i,dev_idle[i]);   {random is a
                        built-in function that returns a random number
                        in the range [0,1)}
         if request[i] then
            begin
            dev_idle[i]:= 0;
            if arbiter_type = '1' then
               begin
               count:= count + 1;
               shuffle[count]:= i     {Store requests}
            end
         end
      end;

if arbiter_type = '1' then      {update request queue if FCFS}
      for i:= count downto 1 do   {Shuffle them}
         begin
         temp:= random(i) + 1;   {Random(i) returns random integer
                                        between 0 and i-1 inclusive     }
         queue[queue_tail]:= shuffle[temp];
         shuffle[temp]:= shuffle[i];
         if queue_tail >= N then
               queue_tail:= 0
         else
               queue_tail:= queue_tail + 1;
         {DEBUG CODE BEGIN}
         if queue_tail = queue_head then
            begin
            writeln (outfile, '*** ERROR: Queue pointers both equal',
                     queue_tail, ' after request for dev', i);
```

```
            stop
          end
          {DEBUG CODE END}
        end
end;   {procedure compute_requests}


procedure next_device (var current_device, last_dev :range);
{
   This procedure determines, given the last device served and the
   requesting device, the device that will next be served.

   The algorithm used for the "biased" arbiter is:

   1) If no device is currently being served, then the lowest numbered
      device making a request, if any, will be the new current device.
   2) If device n is currently being served, then
      a) if 1 or more device numbered n+1 to N (inclusive) is making a
         request, the lowest numbered of these will be served, else
      b) if no device numbered n+1 to N is making a request and 1 or
         more device numbered 1 to n (inclusive) is making a request,
         the lowest numbered of these will be served.

   Note 1: This system is "fair" when the system is busy.  When it is
      idle, it gives a slight advantage to lower numbered devices in the
      event that 2 or more devices simultaneously make a request.
   Note 2: The "state" of the system is specified completely by a) the
      vector of pending requests and b) the device currently being
      served.

   The algorithm used for the "fair" arbiter (added 21 June, 1984) is:

   1) If device n is currently being served (or was the last to be served
      in the case where no device is currently being served) then
      a) if 1 or more device numbered n+1 to N (inclusive) is making a
         request, then the lowest numbered of these will be served, else
      b) if no device numbered n+1 to N is making a request and 1 or
         more device numbered 1 to n (inclusive) is making a request then
         the lowest numbered of these will be served.
   2) On start-up, when there is no "last-served" device, and 2 or more
      devices simultaneously make requests, the lowest numbered one will
      be served.  Note that this can only occur for the very first
      request in the simulation.

   For the "FCFS" arbiter (added 6 July, 1984) a queue of requesting
   devices is maintained; the procedure merely picks off the top
   element in the queue and updates the queue pointers,

   Inputs: N, current_dev, request, queue, queue_head, queue_tail
   Outputs: current_dev, last_dev, queue, queue_head
}
var  temp, count    :range;
     i              :integer;
```

```
begin
if arbiter_type = '1' then
   begin
   if queue_head = queue_tail then    {empty--no pending requests}
      current_dev:= 0
   else    {get topmost element in queue and update pointers}
      begin
      current_dev:= queue[queue_head];
      if queue_head >= N then
            queue_head:= 0
      else
            queue_head:= queue_head + 1;
      last_dev:= current_dev
   end
   end
else   {'F' or 'B' arbiter}
   begin
   if arbiter_type = 'F' then
                              i:= last_dev
                           else
                              i:= current_dev;
   temp:= 0; count:= 0;
   repeat
      i:= i + 1; if i=N+1 then i:= 1;
      count:= count + 1;
      if request[i] then temp:= i
   until (temp=i) or (count=N);
   current_dev:= temp;
   if temp <> 0 then last_dev:= temp
end
end;   {procedure next_device}


procedure update;
{
   This procedure updates all variables that need to be updated after
   every simulation step, for keeping track of various parameters.

   Updated variables (i.e., input and output):
   clock, request, dev_wait, dev_idle, idle_time, busy_time, n_busy,
   n_idle, sum_busy, sum_idle, sum_busy_sq, sum_idle_sq, busy_pdf,
   idle_pdf, wait_pdf, dev_req, longest_busy, longest_idle, n_cust_pdf

   Other inputs: N, MAX_BUSY, MAX_IDLE, current_dev
   Other outputs: n_cust
}
var  i, temp   :integer;

begin
clock:= clock + 1;

{Compute number of customers}
n_cust:= 0;
```

```
for i:= 1 to N do
      if request[i] then
         begin
         n_cust:= n_cust + 1;
         dev_wait[i]:= dev_wait[i] + 1
         end
      else
         dev_idle[i]:= dev_idle[i] + 1;
n_cust_pdf[n_cust]:= n_cust_pdf[n_cust] + 1;

{Do busy and idle times now}
if n_cust = 0 then   {no customer to be served}
   begin
   {DEBUG CODE BEGIN}
   if current_dev <> 0 then
      begin
      writeln (outfile, '*** ERROR: n_cust =', n_cust, ' but current_dev =',
               current_dev);
      stop
   end;
   {DEBUG CODE END}
   if idle_time = 0 then   {Just became idle}
      begin
      {DEBUG CODE BEGIN}
      if (busy_time = 0) and (clock <> 1) then
         begin
         writeln (outfile, '*** ERROR: idle_time & busy_time both 0',
                  ' at clock =', clock);
         stop
      end;
      {DEBUG CODE END}
      if busy_time <> 0 then   {false at clock=1 only}
         begin
         n_busy:= n_busy + 1;
         sum_busy:= sum_busy + busy_time;
         sum_busy_sq:= sum_busy_sq + sqr(busy_time);
         if busy_time >= longest_busy then longest_busy:= busy_time;
         if busy_time <= MAX_BUSY then
            begin
            temp:= round(busy_time);
            busy_pdf[temp]:= busy_pdf[temp] + 1
            end
         else
            busy_pdf[0]:= busy_pdf[0] + 1
      end;
      busy_time:= 0
   end;
   idle_time:= idle_time + 1;
   {DEBUG CODE BEGIN}
   if round(busy_time) <> 0 then
      begin
      writeln (outfile, '*** ERROR: busy_time =', busy_time,
                ' during idle period');
```

```
      stop
   end
   {DEBUG CODE END}
   end    {"if n_cust=0"}
else   {n_cust <> 0 --- customers to be served}
   begin
   {DEBUG CODE BEGIN}
   if current_dev = 0 then
      begin
      writeln (outfile, '*** ERROR: n_cust =', n_cust, ' but current_dev =',
               current_dev);
      stop
      end
   else if not request[current_dev] then
      begin
      writeln (outfile, '*** ERROR: request[', current_dev:2, '] = false');
      stop
   end;
   {DEBUG CODE END}
   if busy_time = 0 then   {just became busy}
      begin
      {DEBUG CODE BEGIN}
      if (idle_time = 0) and (clock <> 1) then
         begin
         writeln (outfile,'*** ERROR: idle_time and busy_time both zero at',
                  ' clock =', clock);
         stop
      end;
      {DEBUG CODE END}
      if idle_time <> 0 then   {false at clock=1 only}
         begin
         n_idle:= n_idle + 1;
         sum_idle:= sum_idle + idle_time;
         sum_idle_sq:= sum_idle_sq + sqr(idle_time);
         if idle_time >= longest_idle then longest_idle:= idle_time;
         if idle_time <= MAX_IDLE then
            begin
            temp:= round(idle_time);
            idle_pdf[temp]:= idle_pdf[temp] + 1
            end
         else
            idle_pdf[0]:= idle_pdf[0] + 1
      end;
      idle_time:= 0
   end;
   busy_time:= busy_time + 1;
   {DEBUG CODE BEGIN}
   if round(idle_time) <> 0 then
      begin
      writeln (outfile, '*** ERROR: idle_time =', idle_time,
               ' during busy period');
      stop
```

```
   end;
   {DEBUG CODE END}

   {Now do waiting times}
   temp:= dev_wait[current_dev];
   wait_pdf[current_dev,temp]:= wait_pdf[current_dev,temp] + 1;
   dev_wait[current_dev]:= 0;
   dev_req[current_dev]:= dev_req[current_dev] + 1;
   request[current_dev]:= false   {service completed}
end   {else clause of "if n_cust = 0"}
end;   {procedure update}


procedure output;
{
   This procedure outputs the result of the simulation.

   Inputs: N, MAX_BUSY, MAX_IDLE, n_cust_pdf, clock, dev_req, wait_pdf,
   sum_busy, n_busy, sum_busy_sq, sum_idle, n_idle, sum_idle_sq,
   longest_busy, busy_pdf, longest_idle, idle_pdf, arbiter_type, p, p1,
   idle_len
}
var  i,j                          :range;
     tot_req, temp, sum, average,
     mean_square                  :real;
     answer                       :char;
     count, limit                 :integer;

begin
writeln (outfile, 'Arbiter type = ', arbiter_type,
       '  Req. prob. (dev 1) =',p1:8:5, '  Req. prob. (dev 2-8) =', p:8:5);
writeln (outfile, 'Guaranteed idle length =', idle_len:4);
writeln (outfile, 'Distribution of number of customers:');
average:= 0;
for i:= 0 to N do
   begin
   temp:= n_cust_pdf[i]/clock;
   writeln (outfile, 'pi[', i:2, '] =', temp);
   average:= average + i * temp
end;
writeln (outfile, 'Average number in system =', average);

writeln (outfile, 'Distribution of waiting times:');
tot_req:= 0;
for i:= 1 to N do tot_req:= tot_req + dev_req[i];
average:= 0; mean_square:= 0;
for j:= 1 to N do
   begin
   sum:= 0;
   for i:= 1 to N do sum:= sum + wait_pdf[i,j];
   temp:= sum/tot_req;
   writeln (outfile, 'w[', j:2, '] =', temp);
   average:= average + j * temp;
```

```
   mean_square:= mean_square + sqr(j) * temp
end;
writeln (outfile, 'Average waiting time =', average, ',   Variance =',
                  mean_square - sqr(average));

repeat
   write (con, 'Do you want waiting-time distributions',
          ' for each device ? ');
   readln (infile, answer); answer:= upcase(answer)  {upcase is built-in
                          function that converts a character to upper case}
until answer in ['Y','N'];
if answer = 'Y' then
      for i:= 1 to N do
         begin
         writeln (outfile, 'Device #', i:2,
                  '    (', dev_req[i]:8:0, ' requests)');
         if dev_req[i] = 0 then
            writeln (outfile, '*** No requests for device', i:2, ' ***')
         else
            begin
            average:= 0; mean_square:= 0;
            for j:= 1 to N do
               begin
               temp:= wait_pdf[i,j]/dev_req[i];
               writeln (outfile, 'w', i:2, '[', j:2, '] =', temp);
               average:= average + j * temp;
               mean_square:= mean_square + sqr(j) * temp
            end;
            writeln (outfile, 'Average =', average, ',   Variance =',
                              mean_square - sqr(average))
         end
      end;

if n_busy <= 0.5 then
   writeln (outfile, 'Less than 1 busy period; cannot compute average')
else
   writeln (outfile, 'Busy time: mean =', sum_busy/n_busy, '   variance =',
          (n_busy * sum_busy_sq - sqr(sum_busy)) / sqr(n_busy));

if n_idle <= 0.5 then
   writeln (outfile, 'Less than 1 idle period; cannot compute average')
else
   writeln (outfile, 'Idle time: mean =', sum_idle/n_idle, '   variance =',
          (n_idle * sum_idle_sq - sqr(sum_idle)) / sqr(n_idle));

repeat
   write (con, 'Do you want busy/idle time distributions ? ');
   readln (infile, answer); answer:= upcase(answer)
until answer in ['Y','N'];
if answer = 'Y' then
   begin
   writeln (outfile, 'Busy time distribution: ');
   if longest_busy >= MAX_BUSY then
```

```
            limit:= MAX_BUSY
         else
            limit:= round(longest_busy);
         for count:= 1 to limit do
               writeln (outfile, 'b[', count:4, '] =', busy_pdf[count]/n_busy);
         writeln (outfile, 'Longest busy period =', longest_busy);
         writeln (outfile, 'Number of busy periods with length >', MAX_BUSY:5,
                  ' is', busy_pdf[0]);

         writeln (outfile, 'Idle time distribution: ');
         if longest_idle >= MAX_IDLE then
            limit:= MAX_IDLE
         else
            limit:= round(longest_idle);
         for count:= 1 to limit do
               writeln (outfile, 'i[', count:4, '] =', idle_pdf[count]/n_idle);
         writeln (outfile, 'Longest idle period =', longest_idle);
         writeln (outfile, 'Number of idle periods with length >', MAX_IDLE:5,
                   ' is', idle_pdf[0])
      end
      end;   {procedure output}

   {
    BEGIN MAIN PROGRAM
   }
   begin
   io;   {Initialize input/output devices}
   repeat
      initialize;
      init_pr_req (idle_len, p1, p);

      repeat
         input (iter_limit, busy_limit);

         repeat
            compute_requests;
            next_device (current_dev, last_dev);
            update;
            if frac(clock/1000) <= 0.0001 then
                  writeln (con, 'Iteration', clock:8:0, ',   Busy periods =',
                           n_busy:8:0)
         until keypressed or (clock >= iter_limit) or (n_busy >= busy_limit);
               {keypressed is built-in function; returns true if any key pressed}

         repeat
            writeln (outfile, '*** Simulation complete ***');
            writeln (outfile, 'Iteration', clock:8:0, ',   Busy periods =',
                              n_busy:8:0);
            write (con, 'Output (O), Continue (C), Restart (R) or',
                           ' Quit (Q) ? ');
            readln (infile, answer); answer:= upcase(answer);
            if answer = 'O' then output
         until answer in ['C','R','Q']
```

```
   until answer in ['R','Q']

until answer = 'Q';
writeln (con, ^G, '*** PROGRAM COMPLETE ***');
close (infile);
close (outfile)

end.    {program Arbiter_simulate}
```